TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OSKARI RUUTIAINEN

SERVERLESS-ARKKITEHTUURIN HYÖDYNTÄMINEN OHJELMISTO-PROJEKTISSA

Diplomityö

Tarkastaja: Prof. Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 31.5.2017

TIIVISTELMÄ

OSKARI RUUTIAINEN: Serverless-arkkitehtuurin hyödyntäminen ohjelmistoprojektissa
Tampereen teknillinen yliopisto
Diplomityö, 52 sivua
Lokakuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Kari Systä
Avainsanat: Serverless, AWS Lambda, Amazon Web Services, Cloud Computing

Ohjelmistoprojektissa yksi tärkeimmistä asioista on valita sopiva arkkitehtuuri rakennettavan järjestelmän pohjaksi. Vääränlainen arkkitehtuuri nostaa kehityksen kustannuksia niin projektin kehitysvaiheessa ennen julkaisua, kuin järjestelmän tuotantojulkaisun jälkeisen jatkokehityksen ja ylläpidon kustannuksia. Serverless-arkkitehtuurissa järjestelmä tai sen osa koostuu pilvipalvelussa ajettavista tilattomista funktioista, joita kutsutaan kun niitä tarvitaan. Funktio, tai joukko funktioita, ajetaan pilvipalvelun tarjoajan ympäristössä, jolle ohjelmistokehittäjän ei tarvitse tehdä monimutkaista konfigurointia. Tällaisen funktion kustannukset koostuvat tavallisesti siitä kuinka usein funktio kutsutaan, kuinka kauan sen ajaminen kestää ja kuinka tehokas laskentayksikkö on konfiguroitu funktiota käyttöön.

Tässä diplomityössä serverless-arkkitehtuuria käsitellään Amazon Web Services Lambda -palvelun avulla ja käytännön esimerkkinä muutetaan pieni verkkosovellus nimeltään Afterwork-alert hyödyntämään serverless-palveluita. Työn puitteissa myös tutustutaan asiantuntijoiden haastatteluiden tasolla Gofore Oy:n toteuttamiin laajempiin ohjelmistoprojekteihin, joissa hyödynnetään serverless-palveluita.

Afterwork-alert-sovelluksen muutosprojektin ja haastattelujen perusteella voidaan todeta serverless-palveluiden olevan hyödyllinen lisä laajojen tietojärjestelmien arkkitehtuurissa. Pienemmissä tietojärjestelmissä tai sovelluksissa serverless-arkkitehtuuri voi toimia jopa ainoana sovellettava arkkitehtuurina.

ABSTRACT

OSKARI RUUTIAINEN: Utilization of serverless-architecture in software project
Tampere University of Technology
Master of Science thesis, 52 pages
October 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Kari Systä
Keywords: Serverless, AWS Lambda, Amazon Web Services, Cloud Computing

One of the most important things in a software project is to choose an appropriate software architecture to depend on. Wrong choise of architecture may increase development costs before and after production release, and also result in large maintanance costs of the software system. When using a serverless architecture, the system or part of the system is composed of stateless functions that are executed in cloud environment, and only called when needed. Function, or a group of function, are executed in cloud environment of service provider. Software developer does not have to do any complicated configuration for that environment. Costs of serverless functions usually consists of how often the function is called, how long does the execution take and how high computational performance is needed for the function.

In this master's thesis serverless architecture is explored with Amazon Web Services Lambdas, and by refactoring a small integration application, called Afterwork-alert, to utilize serverless architecture. Thesis also goes through interviews with two software professionals from Gofore Oy to get to know how serverless services are used in large scale software projects.

Based on the refactoring project of the Afterwork-alert application and professional interviews, one can state that serverless services are beneficial as a part of the architecture of a large scale information system. For a small scale systems or projects serverless architecture can even be only applied architecture.

ALKUSANAT

Tietotekniikka alkoi kiinnostaa minua aikoinaan lähinnä pelien kautta. Muistan edelleen hyvin ensimmäisen itse kootun pelitietokoneen hankinnan. Kiitos veljelleni Jarnolle avustamisesta tuon kyseisen tietokoneen kokoamisessa, sekä muutoin tietotekniikan alalla esimerkkinä olemisesta ja hyvistä keskusteluista tietotekniikasta vuosien varrella. Lapsuudessa koulunkäynnin tärkeyden painottamisesta isot kiitokset vanhemmilleni. Kiitos myös muille perheenjäsenilleni ja ystävilleni opintojeni tukemisesta. Lisäksi haluan mainita erikseen Maijan, joka heilutti erittäin ansiokkaasti punakynää diplomityöni oikolukemisessa. Tärkeimpänä kuitenkin haluan kiittää vaimoani Annukkaa kanssani Tampereelle muuttamisesta opintojeni alussa, sekä jatkuvasta ja väsymättömästä tuesta arjessa ja opinnoissa, mukaan lukien diplomityön viimeisteleminen.

Opintojen alussa tavoitteeni oli valmistua viidessä vuodessa. Tavoite tietenkin muuttui sitä mukaa kun tein töitä opintojen ohella. Siitä huolimatta etten löytänyt työn puolesta asiakasprojekteista soveltuvaa diplomityöaihetta, haluan kiittää työnantajaani Gofore Oy:ta joustavuudesta ja tuesta diplomityön tekemisessä. Erityiskiitos haastatteluihin osallistuneille Joni Ollikaiselle ja Jari-Pekka Voutilaiselle. Lisäksi kiitos myös diplomityössä käsitellyn esimerkkisovelluksen kehittämiseen kanssani osallistuneelle Joni Laurilalle inspiraation tarjoamisesta AWS Lambdojen pariin. Kiitos myös professori Kari Syställe diplomityöni ohjaamisesta, palautteesta, tarkastamisesta, sekä antoisista keskusteluista pilvipalvelujen parista.

Tampereella 24.10.2017

Oskari Ruutiainen

SISÄLLYS

TERMISTÖ

Afterwork-alert	Serverless-palveluita hyödyntäväksi muunnettava sovellus
AWS	Amazon Web Services
AWS CloudWatch	Amazon Web Services CloudWatch metriikan keräämisen ja toimintojen ajastamisen palvelu
AWS EC2	Amazon Web Services Elastic Compute Cloud on palvelu, joka tarjoaa virtuaalipalvelimia
AWS IAM	Amazon Web Services Identity and Access Management
AWS Lambda	Amazon Web Services Lambda
AWS S3	Amazon Web Services S3 tietovarasto
git	Versiohallintasovellus
GitHub	Git-versiohallinnan ja hallintatyökalut tarjoava pilvipalvelu
Gofore Oy	Asiantuntijahaastattelujen kohdeyritys
Heroku	PaaS-palveluntarjoja
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
FaaS	Function as a Service
Serverless	Arkkitehtuuri ilman dedikoituja palvelimia
Serverless Framework	Pilvi-infrastruktuurin rakentamiseen soveltuva työkalu
Slack	Ryhmäkeskustelun organisaatiolle tarjoava pilvipalvelu
Untappd	Juomien arvosteluun soveltuva sosiaalisen median sovellus

1. JOHDANTO

Tässä diplomityössä tarkastellaan serverless-arkkitehtuureja, eli Function-as-a-Service-palveluita ja niiden kanssa hyödynnettäviä muita palveluita. Function-as-a-Service lyhennetään yleisesti FaaS-akronyymillä. Lisäksi työssä vertaillaan miten sovelluksen rakennetta ja käytettyä ajoympäristöä täytyy muuttaa siirryttäessä perinteisestä virtuaalikoneympäristöstä serverless-palveluiden käyttöön. Työssä tutustutaan esimerkkiprojektin kautta tarkemmin Amazon Web Services Lambda -palveluun, joka on parhaiten tunnettu FaaS-palvelu, sekä muihin saman palveluntarjoajan palveluihin, joita käytetään usein yhdessä Amazon Web Services Lamdojen, eli AWS Lambdojen kanssa. Työssä esitellään myös muita palveluntarjoajia dokumentaatioiden tasolla vertaillen. Lisäksi käsitellään pintapuolisesti muita tietojärjestelmäarkkitehtuureja, joiden toteuttamisessa serverless-palveluita voi hyödyntää.

Diplomityön esimerkkisovelluksessa hyödynnettävä Untappd on juomien arvosteluun suunnattu sosiaalisen median verkkopalvelu ja mobiilisovellus. Toinen esimerkkisovelluksessa hyödynnettävä sovellus on Slack, joka on tiimiviestintään suunnattu verkkopalvelu. Diplomityössä esitellään Untappd- ja Slack-palvelut yhdistävä Afterwork-alert-sovellus ja käsitellään esimerkkinä sovelluksen muutosprojekti perinteisemmältä pilvipalvelinalustalta Amazon Web Services Elastic Compute Cloud, eli AWS EC2-instanssilta käyttämään AWS Lambdoja muiden tarpeellisten palveluiden kanssa. Projektista pyritään tunnistamaan selkeät hyödyt ja haasteet ratkaisuineen serverless-palveluiden käytössä. Lisäksi työssä haastatellaan Gofore Oy:n asiantuntijoita serverless-palveluiden käytöstä ohjelmistoprojekteissa. Haastatteluissa havaittuja hyötyjä ja haittoja serverless-palveluiden hyödyntämisestä ohjelmistoprojekteissa vertaillaan esimerkkinä käytetyn muutosprojektin havaintoihin. Lisäksi tarkastellaan erilaisia palveluntarjoajien dokumentaatioita ja hinnastoja kustannusten muodostumisen ymmärtämiseksi.

Havaintojen perusteella pyritään tunnistamaan erilaiset tilanteet, joissa serverless-palveluiden hyödyntäminen ohjelmistoprojektissa on taloudellisesti tai teknisesti hyödyllistä, sekä havaitsemaan ne tilanteet joihin serverless-palvelut eivät sovellu. Havaintojen perusteella määritellään myös millaisia haasteita perinteisen pilvipalvelimella tai omalla palvelimella ajettavan sovelluksen muuttaminen täysin FaaS-palveluiden varaan rakennetuksi tuo ja miten näitä ongelmia voidaan ratkaista.

Luvussa 2 esitellään pilvipalveluita ja erilaisia käytettyjä arkkitehtuureja teoriatasolla. Luvussa 3 tutustutaan serverless-palveluihin ja niiden käyttöä tukeviin palveluihin. Lisäksi esitellään pilvi-infrastruktuurin rakentamiseen tarkoitetun Serverless Framework -työkalun käyttöä. Luvussa 5 tutustutaan esimerkkinä käytettyyn sovellukseen, joka on kehitetty pilvialustalle varsin perinteisenä Node.js-sovelluksena. Luvussa 6 esitetään kuinka edellisen luvun sovellus muutettiin toimimaan ilman jatkuvasti käynnissä olevaa palvelinta tai virtuaalipalvelinta, hyödyntämällä AWS Lambdoja ja siihen liitettäviä muita pilvipalveluita. Luvussa 7 kuvataan asiantuntijahaastattelut serverless-palveluiden käytöstä ja havainnot haastatteluista. Luku 4 käsittelee eri serverless-palveluntarjoajien vertailua teknisiltä ominaisuuksiltaan dokumentaation perusteella. Luvussa 8 analysoidaan hyötyjä ja haittoja työssä läpi käydyistä aiheista. Luku 9 toimii yhteenvetona diplomityölle.

2. PILVIPALVELUT JA -ARKKITEHTUURIT

Tässä luvussa käsitellään perinteisiä pilvipalveluiden jaotteluja ja ominaispiirteitä. Lisäksi käsitellään mikropalvelu- ja monikerrosarkkitehtuurien pääpiirteet, joiden toteuttamiseen voi hyödyntää tässä diplomityössä käsiteltäviä pilvipalveluita. Monikerrosarkkitehtuurit käsitellään yleisellä tasolla kolmikerrosarkkitehtuurin avulla. Mikropalveluista ja serverless-arkkitehtuureista käytetään esimerkkinä Amazonin palveluita.

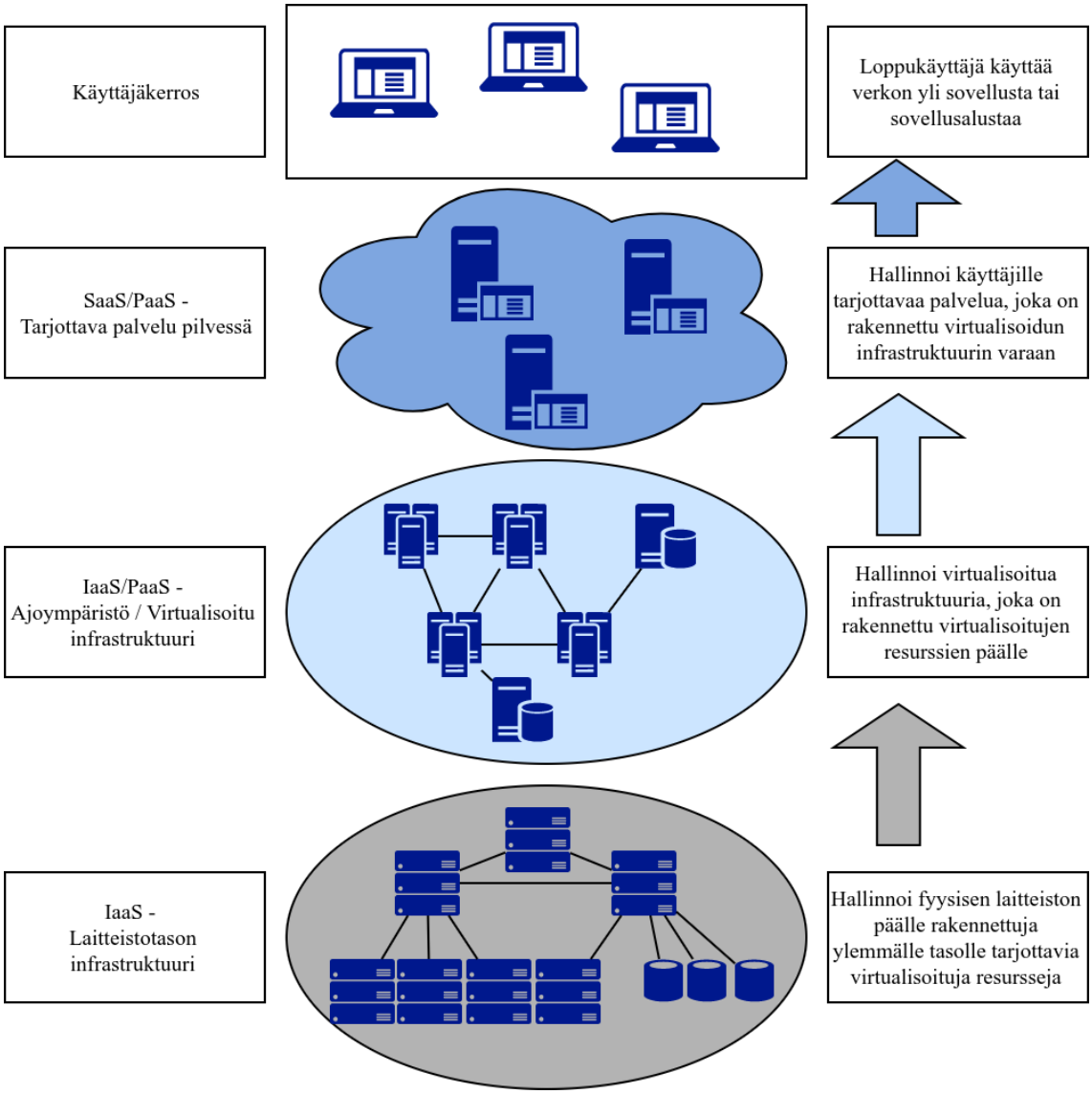
Pilvipalveluiksi voidaan määritellä kaikki verkon yli tarjottavat palvelut. Näitä ovat esimerkiksi laiteinfrastruktuurin, sovelluksen ajoympäristön ja kehitysalustan tarjoaminen tai valmiin verkkopalveluna toteutetun sovelluksen tarjoaminen verkon ylitse asiakkaalle.

2.1 Pilvipalvelujen jaottelut

Yleisellä tasolla pilvipalvelut on jaoteltu kolmeen luokkaan, jotka ovat IaaS (Infrastructure as a Service), PaaS (Platform as a Service) ja SaaS (Software as a Service) [1]. IaaS-tason palvelut sisältävät laitteistotason palvelut, kuten palvelinlaitteiston, laskentatehon ja tallennustilan tarjoamisen asiakkaalle verkon yli [2]. Palvelut voivat olla virtualisoituja tai asiakkaalle varattuja fyysisiä palvelimia [2].

Kuvassa 2.1 IaaS-palvelut sijoittuvat kahdelle alimalle kerrokselle. Esimerkki IaaS-palvelusta on Amazon Elastic Compute Cloud eli Amazon EC2 [3]. EC2-instanssi on käytännössä monipuolinen virtualisoitu palvelin. EC2-palvelinta käyttäessä tilaajalle jää erittäin vapaat kädet käyttöjärjestelmän valinnasta alkaen, mutta toisaalta myös ylläpitovastuu on tällöin tilaajalla. EC2-instanssia ylläpitävä taho voi olla eri kuin sovelluksen kehittävä taho. Tällöin ylläpitävän tahon ja sovelluskehittäjän välillä on tiivis yhteistyö päivityksien ajoittamisessa ja muissa ajoympäristöihin liittyvissä kysymyksissä. IaaS-palveluksi voidaan laskea myös dedikoitujen virtualisoimattomien palvelimien tarjoaminen ja hallinnointi asiakkaan puolesta [4].

SaaS-palvelut ovat loppukäyttäjille suunnattuja, yleensä verkkoselaimen kautta saavutettavia verkkopalveluita, jotka usein korvaavat omalle paikalliselle koneelle asennettavia ohjelmistoja [2]. Esimerkkinä näistä sovelluksista ovat esimerkiksi Spotify ja useat



Kuva 2.1 *Pilvipalveluiden jakaminen kolmeen eri abstraktiotasoon [4]*

Googlen palvelut kuten Drive, Sheets ja Slides. Kuvassa 2.1 SaaS-palvelut sijoittuvat toiseksi ylimmälle kerrokselle, niiden yllä ollen vain käyttäjäkerros, jotka ovat usein selaimessa toimivat asiakassovellukset [4].

PaaS-tason palvelut ovat sovelluskehittäjille suunnattuja kehitysalustoja, jotka tarjoavat valmiin laite- ja sovellusympäristön kehittäjän tarpeisiin [1]. Kuvassa 2.1 PaaS-palvelut sijoittuvat kahdelle keskimmäiselle kerrokselle. PaaS-palvelu voi toimia virtualisoidun infrastruktuurin varassa tai se voi toimia suoraan laitetason palveluiden varassa. Molemmissa tapauksissa PaaS-palvelu tarjoaa ylemmälle SaaS-tason palveluille virtualisoidun valmiin ajoympäristön [4]. Näiden avulla sovelluskehitystyö helpottuu oleellisesti ympäristön ylläpidon osalta. Hyvä esimerkki PaaS-palveluntarjoajasta on Heroku, joka toimii Amazonin tarjoaman IaaS-ympäristön varassa, tarjoten PaaS-

ympäristön loppukäyttäjälle [5]. PaaS-palveluita käytettäessä SaaS-palveluntarjoajan liiketoimintamalli usein eroaa IaaS-palvelun päälle rakennetun SaaS-palvelun liiketoimintamallista [1]. SaaS-palveluntarjoaja pystyy helposti julkaisemaan sovelluksensa PaaS-alustalle, ilman raskasta oman fyysisen palvelimen hankintaa tai pilvipalvelimen tilaamista ja konfigurointia alusta asti [2]. Sovelluskehittäjä pääsee hyödyntämään PaaS-palveluntarjoajan ympäristöön helposti liitettäviä lisäpalveluita, kuten esimerkiksi tietokantaa, logituspalvelua ja verkkoliikenteen monitorointipalvelua. Sovelluskehittäjä voi myös luottaa PaaS-alustan olevan ylläpidetty [2]. PaaS-palveluita käytettäessä liiketoimintamallissa loppukäyttäjä maksaa SaaS-palveluntarjoajalle, joka sitten maksaa PaaS-palveluntarjoajalle kustannukset alustan tarjoamisesta [1].

Näiden lisäksi käytetään paljon erilaisia “as a service”-lyhenteitä eri pilvipalvelutasoista, jotka ovat tarkennuksia yllä listattuihin kolmeen perusmääritelmään [6]. Esimerkkejä sellaisista ovat mm. BaaS – Backend as a Software, DaaS – Database as a Service sekä FaaS – Function as a Service [6]. Luvussa 3 käsitellään tarkemmin viimeisenä mainittuja FaaS-palveluita ja niiden hyödyntämistä.

2.2 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuuri koostuu joukosta pieniä itsenäisiä palveluita, jotka yhdessä muodostavat isomman järjestelmän. Jokainen itsenäinen mikropalvelu toteuttaa oman pienen roolinsa mikropalveluarkkitehtuurissa ja sen toiminnallisuudet rajataan tiukasti siihen. Mikropalvelu toteuttaa oman tehtävänsä hyvin ottamatta kantaa muiden mikropalvelujen rooliin kokonaisuudessa.

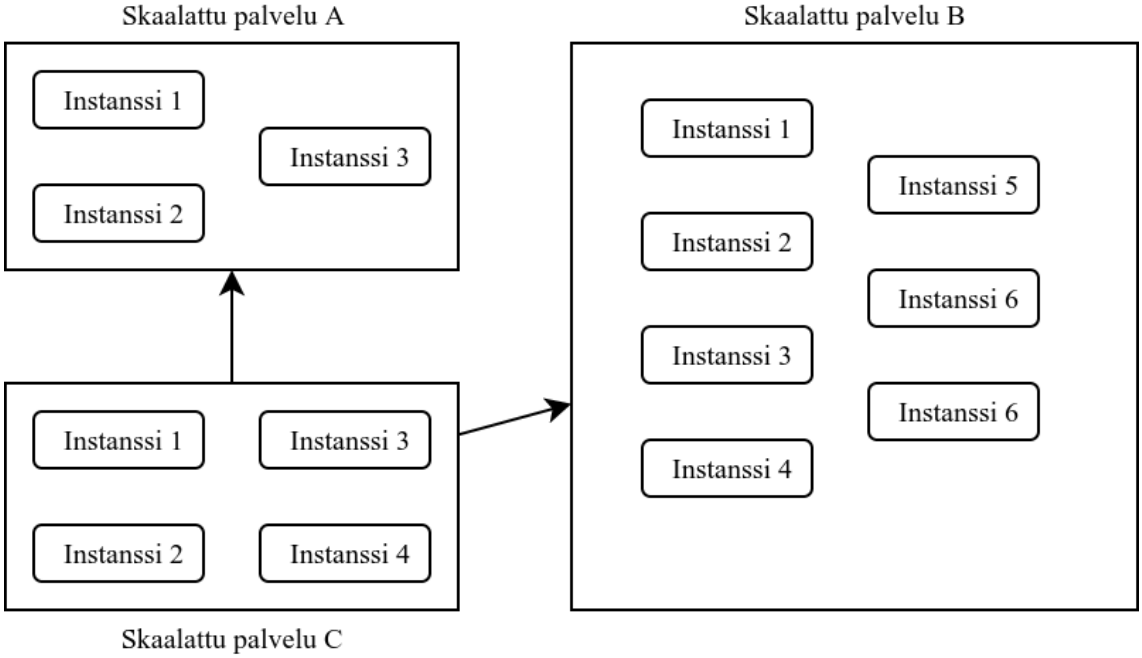
On ongelmallista yrittää määrittää mikropalvelun koodirivimäärälle ylärajaa ohjesäännöksi, koska jo eri ohjelmointikielten erilaiset abstraktiotasot vaikuttavat koodirivien määrään. Nyrkkisäännöksi mikropalvelun määrittelemiseksi on sanottu, että se on jotain sellaista, jonka voi uudelleenkirjoittaa kahdessa viikossa. [7]

Mikropalveluarkkitehtuurissa viestiliikenne palveluiden välillä tapahtuu yleensä tietoverkon ylitse. Tämä edesauttaa palveluiden pitämistä erillisinä ja riippumattomina toistaan [7]. Jokainen palvelu voidaan julkaista toisistaan erillisenä ilman vaatimuksia muista palveluita. Mikropalvelut tarjoavat muille palveluille ohjelmallisen rajapinnan, eli APIn (Application Programming Interface), jonka kautta viestintä tapahtuu [7]. Kuvassa 2.2 havainnollistetussa arkkitehtuurissa on kolme mikropalvelua, joista palvelu C kutsuu verkon yli palveluita A ja B.

Mikropalvelut mahdollistavat myös eri ohjelmointikielten käytön monipuolisesti [7]. Jokainen erillinen mikropalvelu voi olla toteutettu eri kielellä [7]. Näin mahdollistetaan

riskien minimointi huonon teknologiavalinnan suhteen [7]. Monoliittisen järjestelmän kehityksen alussa tehtyjä teknologiavalintoja on huomattavasti vaikeampaa vaihtaa, kuin toteuttaa seuraava pieni mikropalvelu sopivammalla teknologialla [7]. Tämän lisäksi vanhan mikropalvelun rinnalle on helpompaa kehittää uusi, nykyteknologioilla toteutettu vanhan korvaava mikropalvelu. Uuden mikropalvelun ollessa valmis, voidaan se vaihtaa vanhan tilalle helposti. Lisäksi koko tietojärjestelmän vikasietoisuus kasvaa, kun ongelmat voidaan rajata parhaimmillaan yhteen mikropalveluun [7]. Tällöin hyvin rakennettu järjestelmä toimii muilta osin, paitsi virheen kohdanneen mikropalvelun toteuttaman ominaisuuden osalta [7]. Kuvan 2.2 esimerkissä palvelut A, B ja C voivat olla toteutettu eri ohjelmointikielillä ilman vaikutuksia muihin palveluihin. Lisäksi poistettaessa palvelu B käytöstä, palvelu C voi edelleen käyttää palvelun A toiminnallisuuksia hyödyksi.

Mikropalveluiden skaalautuvuus on myös tehokasta [7]. Järjestelmässä voidaan kasvattaa kapasiteettia vain niille palveluille joita on tarve kasvattaa, ilman turhaa kapasiteetin kasvattamista kokonaiselle monoliittiselle järjestelmälle [7]. Kuvassa 2.2 on havainnollistettu eri mikropalvelujen skaalaaminen. Kuvan esimerkissä palvelu C käyttää palveluja A ja B. Palvelulta A vaaditaan vähiten kapasiteettia ja palvelulta B eniten, joten rinnakkaisia instansseja samasta palvelusta on eri määriä käytössä. Palvelun skaalaaminen vaatii luonnollisesti jonkinlaisen mekanismin hoitamaan verkon yli tulevia pyyntöjä eri instansseille palvelusta. Lisäksi jos palvelulla on jonkinlainen tila, täytyy ratkaista tilanhallinta ja mahdolliset rinnakkaisuudesta aiheutuvat ongelmat.



Kuva 2.2 *Jokaista mikropalvelua voi skaalata tarpeen mukaan*

Yksittäisen mikropalvelun julkaisu ja ongelmatilanteessa vanhan version palautus on

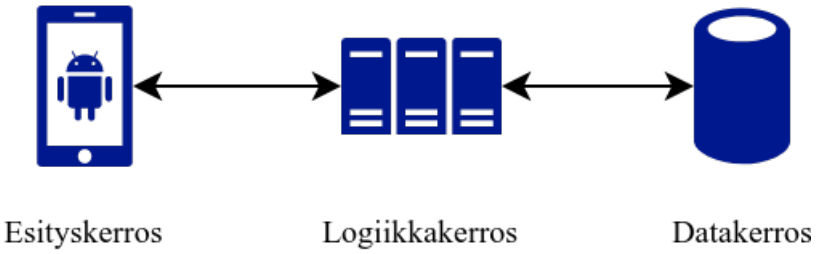
myös mikropalveluarkkitehtuurissa helppoa [7]. Yksittäisen palvelun muutokset ovat irrallisia muista palveluista, joten ongelmat rajautuvat kyseiseen palveluun [7]. Ero vanhan ja uuden yksittäisen mikropalvelun koodissa on aina suhteellisen pieni, koska palvelut itsessään ovat pieniä [7]. Tällöin mahdollisuuksia virheille on myös vähemmän yksittäisessä mikropalvelussa [7]. Nämä seikat mahdollistavat sovelluksen uuden version julkaisemisen usein ja nopeasti [7]. Kuvan 2.2 esimerkissä palvelu A voidaan julkaista täysin erillään palvelusta B. Jos uusi julkaisu ei toimi, voidaan helposti palauttaa edellinen versio. On myös mahdollista asettaa palvelusta A toinen versio yhtäaikaisesti rinnakkain ajoon ja ohjata liikenne uuteen versioon vanhan sijaan.

Mikropalveluarkkitehtuuri mahdollistaa myös kohtuullisen kokoiset kehitystiimit yksittäisten palveluiden parissa, sekä koodipohjan pysymisen järkevässä ja hallittavassa koossa [7]. Eri palveluiden kehitystiimit voivat jopa toimia hyvin erillään toisistaan [7]. Lisäksi mikropalveluarkkitehtuuri mahdollistaa organisaation tietojärjestelmien jatkokehityksen helpommin [7]. Esimerkiksi monoliittinen pitkään kehitetty tietojärjestelmä on erittäin vaikea vaihtaa tai jatkokehittää, koska koodipohja on erittäin laaja ja vaikeasti haltuunotettavissa [7]. Mikropalveluarkkitehtuurin avulla voidaan vaihtaa yksi tietojärjestelmän mikropalveluista kerrallaan. Siten hiljalleen muutetaan koko järjestelmä uudeksi ja soveltumaan uusiin tarpeisiin [7]. Kuvan 2.2 esimerkissä palveluita A, B ja C voidaan kehittää vaikka eri puolella maailmaa omissa kehitystiimeissään. Tiimien välillä tulee olla kuitenkin hyvä kommunikaatio ja kunkin palvelun rajapinnat dokumentoitu riittävällä tasolla.

2.3 Kerrosarkkitehtuuri

Yksinkertainen kolmikerrosarkkitehtuuri sisältää esityskerroksen, logiikkakerroksen ja datakerroksen kuvan 2.3 mukaisesti. Kolmikerrosarkkitehtuurissa esityskerroksella on sovelluksen käyttöliittymä, logiikkakerroksella jonkinlainen sovelluspalvelin tai muu sovelluksen logiikkaa ja tilaa käsittelevä osa sovelluksesta, ja viimeisenä datakerroksella sijaitsee sovelluksen tiedon tallennus tietokantaan ja sen abstrahointi logiikkakerrokselle [8]. Kerrokset ovat loogisesti irrallisia toisistaan, ja ne ovat myös fyysisesti erillään toisistaan.

Kolmikerrosarkkitehtuuri erottelee eri tason sovellukset toisistaan ja mahdollistaa yksittäisen kerroksen sovelluksen päivittämisen ilman, että se vaikuttaa muilla tasolla toimiviin sovelluksiin. Esimerkiksi päivittäessä logiikkakerroksen sovellusta verkkopalvelimella, ei sen vuoksi tarvitse päivittää jokaista mobiililaitteessa toimivaa selainsovelluksena tarjottavaa käyttöliittymää. Jos datakerros on erotettu hyvin logiikkakerroksesta, voidaan vaihtaa järjestelmän tietokannan tarjoava taho toiseen, ilman että se vaikuttaa logiikkakerrokseen. Lisäksi abstraktin tietokantarajapinnan ansiosta logiikkakerroksen kehitystyö



Kuva 2.3 *Yksinkertainen kolmikerrosarkkitehtuuri [8]*

on tehokkaampaa, kun vaivalloinen tietokantakohtainen datan käsittely on abstrahoitu. [8]

Esityskerroksella voi toimia myös useampia erilaisia käyttöliittymiä, jotka kaikki käyttävät samaa logiikkakerrosta. Toisaalta samaa tietokantaa voi käyttää useampia erilaisia logiikkakerroksia, jotka ovat osana eri sovelluksia loppukäyttäjän näkökulmasta.

Kolmikerrosarkkitehtuuri on vain yksi esimerkki kerrosarkkitehtuureista. Yleisemmin voidaan käsitellä monikerrosarkkitehtuureja, joissa eri kerrosten erottelu on tehty vielä hienojakoisemmin. Tämän diplomityön puitteissa kolmikerrosarkkitehtuuria hyödynnetään esimerkkinä arkkitehtuurista, joka voidaan toteuttaa serverless-arkkitehtuurin ja serverless-palveluiden avulla.

3. PALVELIMETON LASKENTA SERVERLESS-PALVELUIDEN AVULLA

Tässä luvussa kuvataan tarkemmin mitä serverless- ja FaaS-palvelut ovat ja miten niitä hyödynnetään. Serverless-palveluista käytetään luvussa 2.1 mainittua FaaS-akronyymia sekä nimeä lambda, jota useat palveluntarjoajat käyttävät FaaS-palveluidensa nimenä [9]. Lambda nimitystä ei tule sekoittaa anonyymeihin funktioihin, joita kutsutaan myös lambdoiksi. Pilvipalvelujen kontekstiin lambda nimitys lienee tullut samankaltaisesta ajattelumallista kuin anonyymeissa funktioissa, mutta käsiteltynä eri abstraktiotasolla ja kontekstissa. Molemmissa käsitellään funktiota, joka ei ole sidoksissa mihinkään muuhun kuin parametreina saamiinsa muuttujiin.

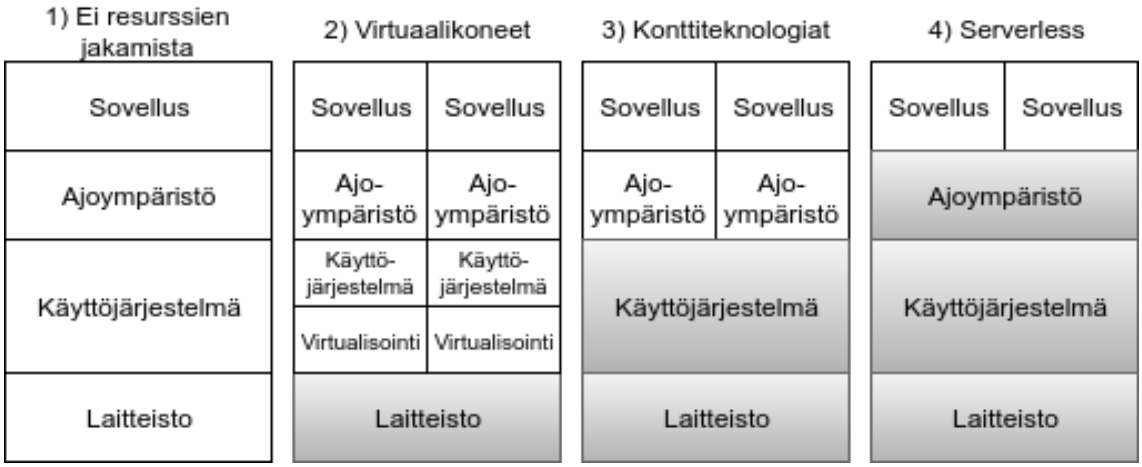
Serverless-palveluista käytetään käytännön esimerkkeinä Amazonin palveluita. Amazonin serverless-palveluita yhdistämällä voidaan toteuttaa esimerkiksi luvussa 2.3 määritellyn monikerrosarkkitehtuurin jokin kerros, tai jokin yksittäinen palvelu luvun 2.2 mukaiseen mikropalveluarkkitehtuuriin [10]. Hyödynnettävistä palveluista esimerkkeinä toimivat Amazon Web Services Lambda [11], joka toteuttaa funktion ajavan FaaS-instanssin, sekä Amazon Web Services Api Gateway [12], joka ohjaa verkko-osoitteeseen tulleen HTTP-pyynnön halutulle AWS Lambdalle. [10]

3.1 Serverless

Laajasti määriteltynä serverless-palvelulla voidaan tarkoittaa kaikkia pilvipalveluita, jotka eivät ole jatkuvasti käynnissä määritellyllä fyysisellä tai virtuaalisella palvelimella ja joissa serverless-palvelut suorittavat tilattoman toiminnon sekä sammuvat tämän jälkeen [10]. Serverless-arkkitehtuuria hyödyntävä tietojärjestelmä voi koostua pelkästään useasta erikseen sarjassa ja rinnakkain kutsuttavasta FaaS-funktiosta, tai jokin osa järjestelmän arkkitehtuurissa on toteutettu käyttäen FaaS-palveluita [13]. Tavallisesti palvelun laskutus on aikaperustaista, eli kustannukset riippuvat siitä kuinka kauan funktiota on ajettu [14]. Lisäksi yleensä laskutukseen vaikuttaa valittu prosessointiteho ja muistin määrä [14]. Funktion suorituksen käynnistävä tapahtuma voi olla esimerkiksi verkkokutsu julkisesta verkosta, toisen pilvipalvelun antama ajastettu heräte tai SaaS-palveluna tarjotun verkkopalvelun käyttöliittymästä annettu komento [9]. AWS Lambdan

laskutus on GB-sekunti perustainen, eli kuinka monta sekuntia funktion suoritus kestää, ja kuinka paljon muistia funktiolle on varattu [14]. Esimerkiksi 128 MB muistimäärän funktiota ajettaessa 3 200 000 sekuntia, täyttyy AWS Lambdan ilmainen ajoaika kuukaudessa, joka on 400 000 GB-sekuntia [14]. Sen jälkeen jokainen suoritus tuottaa käyttäjälle kustannuksia.

Useat serverless-palveluita tarjoavat alustat, kuten Amazon Web Services sekä OpenLambda, ovat nimenneet FaaS-palvelut Lamdboiksi [11] [15]. Lambdat ovat yksi abstraktiotaso lisää virtualisoiduissa ympäristöissä. Kuvasta 3.1 havaitaan millaisia eroja eri tason virtualisoinnissa ja järjestelmän abstrahoinnissa on. Kuvan kohdassa 1 esitetään perinteisen dedikoidulla palvelimella ajettavan sovelluksen ympäristö. Tämän kaltaisessa tilanteessa kaikki osat, fyysinen laitteisto, käyttöjärjestelmä, ajoympäristö ja ajettava sovellus ovat kehittäjän ylläpitämällä palvelimella. On tietenkin mahdollista että tälläisessä ympäristössä on useita sovelluksia samalla palvelimella, mutta tällöin ajoympäristön, käyttöjärjestelmän tai laitteiston päivittämisestä johtuvat käyttökatkokset vaikuttavat moneen sovellukseen.



Kuva 3.1 *Resurssien jakamisen kehittyminen. Harmaat alueet ovat jaettuja resursseja [15]*

Kuvan 3.1 kohdassa 2 kuvataan virtuaalisella palvelimella ajettava sovellus. Tällöin fyysinen laitteisto on yhteinen useammalle virtuaaliselle ympäristölle, joissa jokaisessa voi ajaa erikseen sovelluksia. Virtuaaliympäristön tarjoama käyttöjärjestelmä voi olla eri vaikka jokaiselle tarjotulle virtuaalikoneelle. Tälläinen rajoittaa päivittämisestä johtuvat häiriöt pienemmälle alueelle, sekä pitää mahdollisesti erilaiset ajoympäristöt toisistaan erillään. Eri sovelluksen taustalla olevan virtuaalikoneen käyttöjärjestelmän päivittäminen ei tuota häiriötä toisen sovelluksen ajoon. Tämä kuitenkin pakottaa jokaisen virtuaalisen ympäristön ylläpitämisen erikseen. Lisäksi laitteiston resurssit jaetaan useammalle virtuaalikoneelle, joiden yhtäaikainen rasitus saattaa tuottaa ongelmia fyysiselle laitteistolle. Lisäksi virtuaalisten ympäristöjen taustalla olevan fyysisen laitteiston päivittäminen aiheuttaa tietenkin käyttökatkon virtualisoiduissa ympäristöissä. [15]

Kuvan kohdassa 3 on esitetty konttiteknologioiden hyödyntäminen. Konttiteknologioita hyödyntäessä eri sovellukset ovat ajossa yhteisellä fyysisellä laitteistolla ja saman käyttöjärjestelmän päällä, mutta sovellus ja sen ajoympäristö on paketoitu konttiin. Kontti voidaan asettaa ajoon kaikilla kyseistä konttiteknologiaa tukevilla käyttöjärjestelmillä, kontin sisällöstä riippumatta. Konttiteknologiat ovat kevyempiä virtualisointiratkaisuja kuin koko käyttöjärjestelmän virtualisointi, eivätkä ne vaadi yhtä paljon ylläpitoa kontin sisällä olevalla ympäristölle. Konttiteknologioiden kohdalla sovelluskehittäjä voi myös jättää palveluntarjoajan vastuulle suuremman osan ympäristön ylläpitämisestä ja ylläpitää itse vain ajoympäristöä ja sovellusta, jotka sijaitsevat kontin sisällä. Konttien avulla ei voida myöskään välttyä alla olevan fyysisen laitteiston tai käyttöjärjestelmän päivättämisen aiheuttamilta katkoksilta, mutta ne on jätetty usein esimerkiksi pilvialustan tarjoajan vastuulle ja niistä johtuvat katkokset ovat hyvin minimaalisia. [15]

Konttiteknologiat ovat hyvin laaja kokonaisuus. Todellisuudessa ne ovat monipuolisempia ja mukautuvampia kuin edellä olevasta yksinkertaistetusta kuvauksesta voisi päätellä. Kontteihin voi paketoida hyvin erilaisia kokonaisuuksia ja kontin pohjana käytetty malli voi todellisuudessa sisältää valmistellun ajoympäristön ja jopa sovelluksen valmiina. Tämän työn laajuudessa ei kuitenkaan käsitellä konttiteknologioita tämän enempää.

Kuvan 3.1 viimeinen osa, eli kohta 4, kuvastaa Lambdojen ympäristön rakennetta. Tälläisessä ympäristössä funktio tai sovellus käynnistetään nopeasti missä tahansa palveluntarjoajan virtualisoidussa ympäristössä, jossa on ennalta määritelty ajoympäristö saatavilla [15] . Tällöin käynnistysajat ovat riittävän nopeita ja virtualisoitu ympäristö on helppo pitää päivitettynä, koska sovellus ei ole jatkuvasti käynnissä. Kuitenkin vertailuna todettakoon, se että Lambda-funktio käynnistyy joka kerta, aiheuttaa sen että yhteenlaskutetut vasteajat ovat pidemmät kuin jatkuvasti käynnissä olevien palveluiden vasteajat [16]. Tällaisen FaaS-funktion kehittäjän ei kuitenkaan tarvitse huolehtia että palvelinlaitteisto, käyttöjärjestelmä tai ajoympäristö ovat päivitettynä, vaan ne osat ympäristöstä jätetään palveluntarjoajan huoleksi [15]. Tällöin sovelluskehittäjän täytyy huolehtia vain omasta sovelluksestaan, joka käynnistyy kutsuttaessa aina valmiiksi ylläpidetyssä ja päivitetyssä ympäristössä [15]. Palveluntarjoajan on helppo myös pitää sovelluksen ympäristö kokonaisuudessaan päivitettynä, koska sovellus voidaan käynnistää aina sillä hetkellä saatavilla olevassa virtualisoidussa kontissa. Jos johonkin yksittäiseen ympäristöön, vaikka kokonaiseen palvelinsaliin tehdään ylläpitotoimia, sovellus käynnistetään kutsuttaessa toissijaisessa palvelinsalissa. Tällöin sovelluksen käyttöön ei tule käytännössä koskaan katkoksia.

Serverless-palvelut ovat yleensä tilattomia, mutta niillä voi käsitellä verkon kautta saatavilla olevaa tietokantaa, mikäli tilatietoa tai muuta dataa täytyy säilyttää [13]. Näissä tilanteissa täytyy kuitenkin muistaa, että yhteys tietokantaan, kuten muihinkin ulkoisiin

palveluihin, alustetaan jokaisen yksittäisen funktion ajon alussa. Jos funktiota ajetaan paljon, voivat sen kustannukset nousta jo yksinään tietokantayhteyden luonnin takia merkittävästi verrattuna esimerkiksi jatkuvasti kontissa ajettavaan sovellukseen.

Vertaillessa serverless-palveluja PaaS-palveluihin, huomataan yksittäisen FaaS-instanssin olevan tavallisesti huomattavasti pienempi kokonaisuus kuin kokonainen PaaS-sovellus. Vaikka usein PaaS-alustat perustuvat myös aikalaskutukseen, niistä on huomattavasti vaikeampi tunnistaa oman järjestelmän eri osien kustannukset kokonaisuudessa. Hyödyntämällä useiden PaaS-palveluntarjoajien kauppapaikoissaan tarjoamia lisäosia, voidaan lisäosien kohdalla erotella tehokkaasti sovelluksen eri osa-alueiden tuottamat kustannukset. PaaS-palveluntarjoaja Herokulla tälläinen lisäosien kauppapaikka on Heroku Add-Ons [17]. Kuitenkin PaaS-alustalle kehitetyn oman sovelluksen kustannukset eri osa-alueiden osalta ovat vaikeampia monitoroida. Vaikka tietojärjestelmä olisi kehitetty mikropalveluista koostuvaksi kokonaisuudeksi, yksittäinen PaaS-alustalla ajossa oleva mikropalvelu on yleensä silti suurehko kokonaisuus verrattuna FaaS-instassilla tavallisesti ajettaviin pieniin yhden yksinkertaisen toiminnon toteuttaviin funktioihin. Pilkkomalla mikropalvelu yhä pienempiin osiin ja rakentamalla koko sovellus serverless-palveluita hyödyntäväksi, voidaan monitoroida kustannuksia funktiotasolla. Tulevaisuudessa serverless-palveluntarjoajat saattavat tarjota PaaS-palveluntarjojien markkinapaikkoja vastaavia valmiiden yksittäisten funktioiden markkinapaikkoja, joista voi ottaa käyttöönsä funktion ja maksaa funktiota hallinnoivalle provisiota jokaiselta käyttökerralta [18].

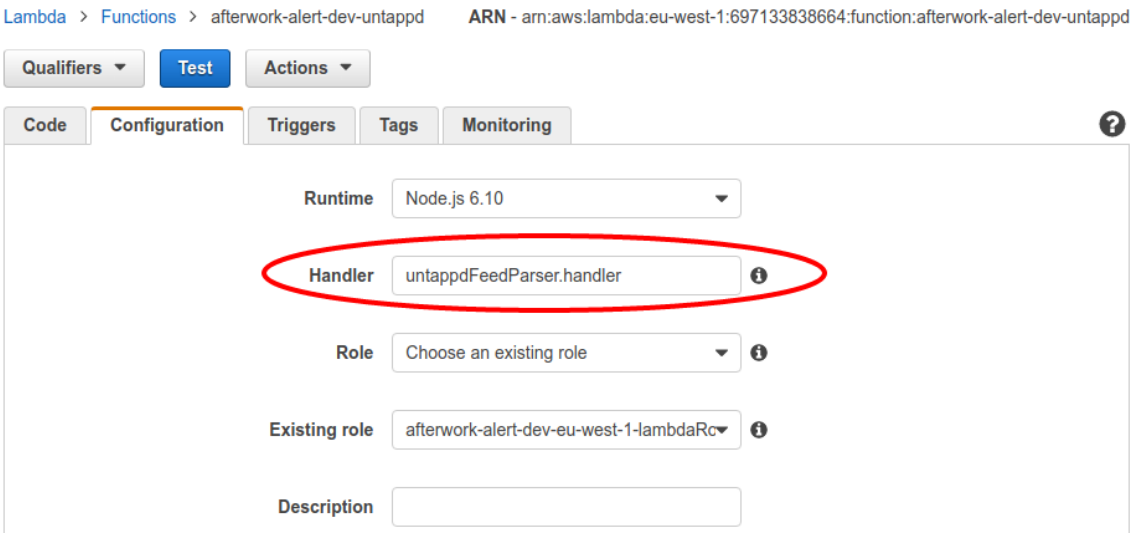
3.2 Amazon Web Services -infrastruktuurin FaaS-palvelu

AWS Lambdat ovat Amazonin versio FaaS-palvelusta. AWS Lambdoilla voi suorittaa tuetulla ohjelmointikielellä tehtyä koodia useiden vaihtoehtoisten ohjelmallisten tapahtumien käynnistämänä. Tuettuja ohjelmointikieliä ovat Node.js (JavaScript), Python, Java 8, ja C# (.NET Core) [11]. Käynnistävinä tapahtumina eli herätteinä voivat toimia useat eri Amazon Web Services -palvelut ja niissä tehtävät toiminnot. Näitä kutsutaan tapahtumalähteiksi tai -herätteiksi. Erilaisia herätteitä ovat esimerkiksi Amazon Simple Storage (Amazon S3) -palveluun tallennettu uusi tiedosto, HTTP-API-kutsu Amazon Web Services Api Gateway kautta tai ajastettu heräte Amazon Web Services Cloudwatch -palvelusta. Luvuissa 3.3 ja 3.4 tarkastellaan tarkemmin mainittuja palveluita.[19]

AWS Lambdan suorituksen käynnistävät herätteet voidaan rajoittaa siten, että vain määritellyt tapahtumalähteet voivat toimia herätteinä. Tällä tavoin voidaan parantaa järjestelmän tietoturvaa. AWS Lambdaan voi määritellä, että ainoastaan yksi tietty Cloudwatch -palveluun asetettu ajastus sallitaan käynnistämään AWS Lambdan, tai että vain

yksi määritelty Api Gateway voi toimia herätteenä ja muiden Api Gatewaylla määriteltyjen HTTP-rajapintojen herätteet eivät aiheuta funktion käynnistymistä. [19]

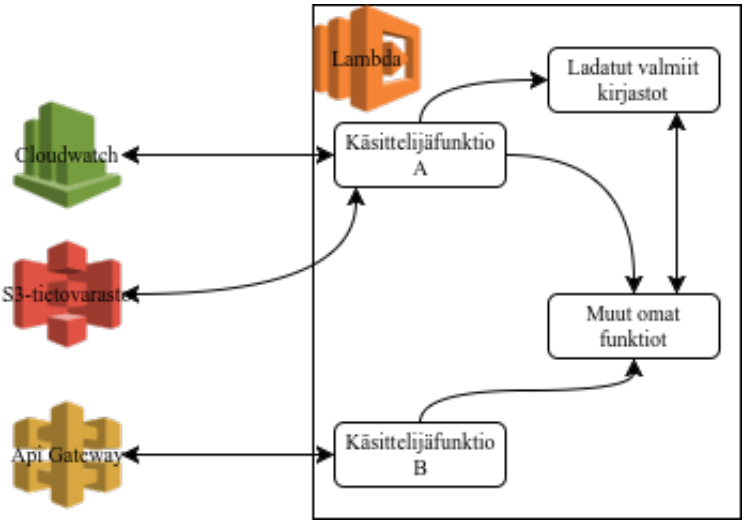
AWS Lambdaan tehdyn sovelluksen ohjelmakoodi koostuu käytännössä käsittelijäfunktiosta, sekä mahdollisesti muista funktioista ja ulkoisista riippuvuuksista, joita käsittelijäfunktio kutsuu. Kaikki Lambdan tarvitsemat lähdekoodit ja kirjastot paketoidaan ja lähetetään Lambdalle suoraan tai S3-tietovaraston kautta. Lambdaan määritellään mikä funktio toimii käsittelijänä, ja samassa julkaisupaketissa voi olla useampia eri lambdojen käsittelyfunktioita. Kuvassa 3.2 on kuvankaappaus AWS Lambda selainkäyttöliittymästä. Punaisella värillä ympyröidyssä kohdassa määritellään mikä funktio lähetetyssä ZIP-pakatussa julkaisupakkauksessa on herätteestä kutsuttava käsittelijäfunktio. Kuvan tapauksessa paketissa on mukana tiedosto nimeltään *untappdFeedParser.js* ja tiedostossa on määritelty *handler* funktio, joka toimii käsittelijäfunktiona. AWS Lambdan selainkäyttöliittymästä voidaan myös testata funktion ajamista *Test*-painikkeesta. Tällöin voidaan määrittää manuaalisesti millaisia parametreja funktiolle antaa testiherätteen yhteydessä, ja siten testata toiminnallisuutta.



Kuva 3.2 *AWS Lambdan käsittelijäfunktion määrittäminen AWS selainkäyttöliittymästä*

Kuvassa 3.3 on havainnollistettu miten eri palvelut voivat tuottaa herätteitä samassa paketoinnissa oleville kahdelle eri Lambda -käsittelijäfunktiolle. Kuvan käsittelijäfunktiot A ja B kutsuvat muita julkaisupakkauksessa mukana olevia funktioita ja mukaan ladattuja ulkopuolisia kirjastoja. Funktiot voisivat kutsua myös täysin AWS Lambdan ulkopuolisia kolmannen osapuolen rajapintoja verkon yli. Ulkopuolisien palveluiden, kuten Untappd ja Slack, käyttöä AWS Lambdan kanssa käsitellään myöhemmin tässä diplomityössä luvussa 5.

AWS Lambdassa ajossa olevalla funktiolla on myös käytettävissään suoritusympäristöön liittyvää informaatiota ajonaikaisesti kontekstioliossa. Funktio voi muun muassa kysyä

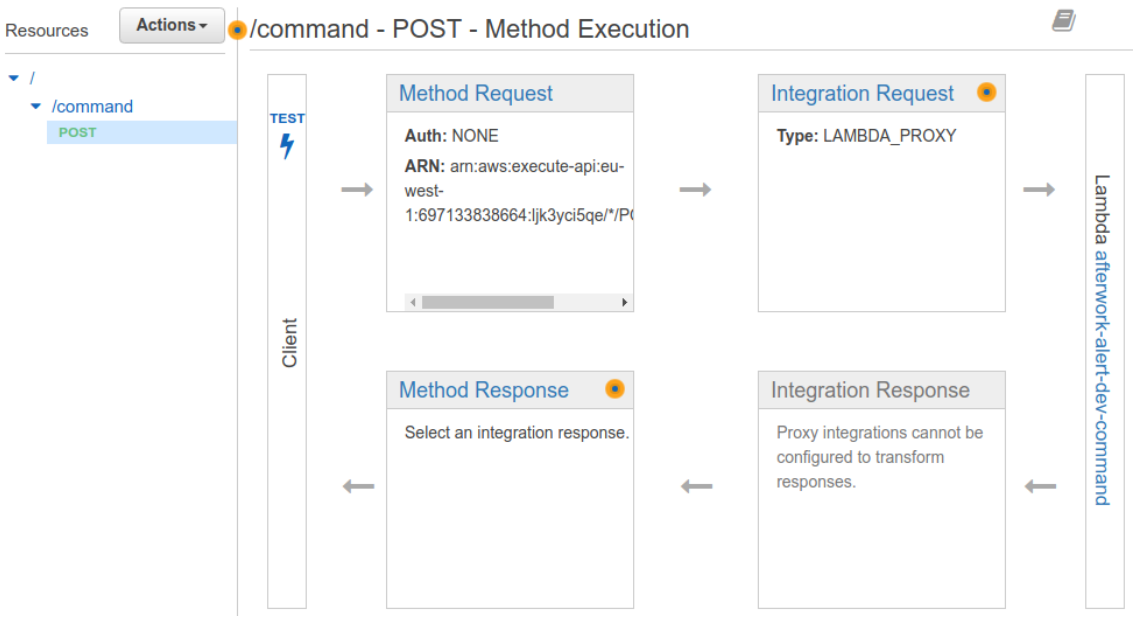


Kuva 3.3 *AWS Lambdan käsittelijäfunktion kutsuminen*

oliolta ennen aikakatkaisua jäljellä olevan ajoajan. Sen ansiosta funktion voi määrittää toimimaan eri tavoin riippuen jäljellä olevasta ajoajasta. Lisäksi kontekstista saa erilaisia Cloudwatch -palvelulla seurattavien lokien yksityiskohtia, Lambdan herättäneen AWS pyynnön ID:n ja mobiilisovelluksen tietoja, jos Lambdaan on liitetty AWS Mobile SDK:n kautta jokin sovellus. AWS pyynnön ID:tä voi hyödyntää ongelmatilanteissa AWS tuen kanssa. [20]

3.3 HTTP-viestinvälitys FaaS-yksikölle Amazon Web Services -infrastruktuurissa

Amazon Web Services Api Gateway -palvelu on tarkoitettu verkkorajapintojen (API:en) määrittämiseen, käyttöönottoon ja hallinnointiin ilman palvelimen ylläpitoa. Asiakassovellukset voivat keskustella APIn kanssa standardeilla HTTPS-viesteillä ja Api Gateway -palveluun voidaan määrittää toiminnallisuuksia niiden perusteella. Api Gatewayn voidaan asettaa toimimaan AWS Lambdalle HTTP-proxynä, eli tällöin Api Gateway vastaanottaa HTTPS-viestit ja luo niiden perusteella herätteen AWS Lambdalle. Api Gateway välittää alkuperäisen HTTP-pyynnön sisällön, eli pyynnön metatiedot, otsaketiedot ja rungon, AWS Lambdalle ja välittää kutsujalle Lambdan antaman vastauksen. Kuvankaappaus esimerkkikonfiguraatiosta AWS selainkäyttöliittymästä on esitetty kuvassa 3.4. Kuvan tapauksessa Api Gateway on määritelty ottamaan POST-tyyppiset HTTP-viestit $/command$ osoitteeseen vastaan ja ohjaamaan ne Lambdalle nimeltä $afterwork-alert-dev-command$. Lambdan vastaus ohjataan takaisin alkuperäiselle kutsujalle. Api Gateway voi myös tallentaa kyselyjä ja vastauksia välimuistiin, joka saattaa pienentää Lambdan ajoaikaa ja siten pienentää kustannuksia. [19]



Kuva 3.4 *Api Gateway konfiguraatio*

Api Gateway on avoin julkiseen verkkoon, mutta se tarjoaa kuitenkin useita erilaisia tietoturvaominaisuuksia. Kaikki hyväksyttävä viestiliikenne on HTTPS-salattua. AWS Lambdalle voi määrittää tarkasti minkä Api Gatewayn luoman API:n yli sen voi käynnistää. Lisäksi Api Gatewayn avulla luodut resurssit saavat kaikki omat yksilölliset resurssinimet (Amazon Resource Name - ARN), joihin voi viitata Amazonin identiteetin ja pääsynhallinnan työkalussa IAM:ssa (Identity and Access Management). Api Gatewaylle voi myös luoda mukautetun autentikaatiofunktion, hyödyntämällä erillistä AWS Lambdaa. AWS Lambdana toteutettu mukautettu autentikaatiofunktio asetetaan palauttamaan IAM pääsynhallinann määrittämä identiteetti kutsujalle, ja siten API Gateway voi määrittää suoritetaanko todellisuudessa HTTP-rajanpinnan kautta kohteena ollutta AWS Lambda -funktiota. [19]

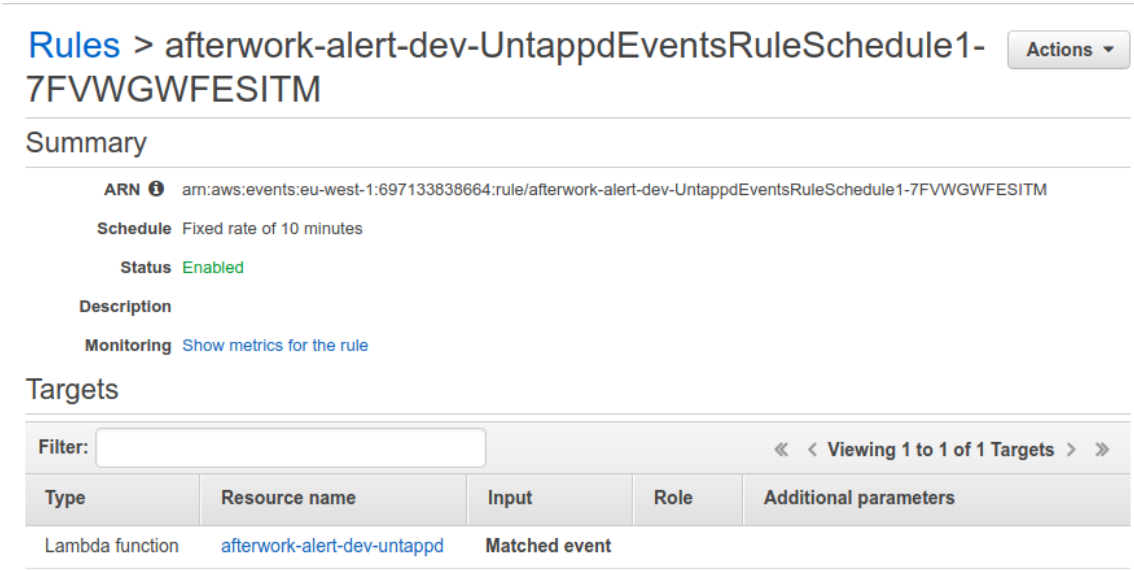
Api Gateway tarjoaa tuen API:n kloonaamisella uudeksi API:ksi, liikenteen rajoittamisen kyselypiikkien hallitsemiseksi ja hyvän monitoroinnin Amazon CloudWatch palvelun avulla. Lisäksi Api Gateway on integroitu Amazon CloudFront -palvelun kanssa siten, että sillä luodut rajapinnat ovat tarjolla useista pääsyverkoista ympäri maailman, joka pienentää kyselyjen vasteaikoja loppukäyttäjälle. [19]

3.4 Ajastettujen toimintojen suorittaminen Amazon Web Services -ympäristöissä

AWS Cloudwatch on palvelu, jonka avulla voi monitoroida lähes mitä tahansa Amazon Web Services ympäristöissä toimivan järjestelmän toimintaa, kerätä metriikkaa, aset-

taa ajastettuja toimintoja sekä reagoida automaattisesti muutoksiin muissa Amazonissa käytössä olevien palveluiden resursseissa. Tämän diplomityön kannalta keskeisin seikka on toimintojen ajastukset. Cloudwatch-palveluun voi määritellä esimerkiksi 10 minuutin välein ajastetun AWS Lambdan kutsumisen halutuilla parametreilla. Cloudwatch voi myös seurata muiden hyödynnettävien palveluiden metriikkaa ja automaattisesti kutsua Lambdaa, skaalata resursseja tai ottaa käyttöön uusia resursseja. [21]

Kuvassa 3.5 on kuvankaappaus AWS CloudWatch -selainkäyttöliittymästä yksittäisestä ajastussäännöstä luvussa 6 käsiteltävästä sovelluksesta. CloudWatch-säännöllä on oma uniikki ARN, eli uniikki Amazon resurssinimi. Resurssinimeen voi viitata muissa palveluissa. Säännölle on määritelty aikataulu, eli kuinka usein tämä toiminto tapahtuu. Lisäksi säännölle voi antaa oman kuvauksen. Targets-osiossa on listatu mihin kaikkiin muihin resursseihin tämä sääntö vaikuttaa, joita tässä tapauksessa on vain yksi AWS Lambda -funktio. Lisäksi näkymästä pääsee kyseisen säännön metriikoihin, joista voi seurata kuinka usein säännön mukainen toiminto on ajettu.



Kuva 3.5 *AWS CloudWatch selainkäyttöliittymä*

3.5 Serverless Framework – Kolmannen osapuolen palvelu

AWS palveluiden päälle rakennetun tietojärjestelmän infrastruktuuria voi hallinnoida myös kolmannen osapuolen palveluiden, kuten Serverless Framework -palvelun avulla. Serverless Framework tarjoaa tuen neljälle isolle serverless-palveluita tarjoavalle alustalle, jotka ovat Amazon Web Servicesin, Microsoft Azuren, IBM OpenWhiskin sekä Google Cloud Platformin. Serverless Framework on lisensoitu avoimen lähdekoodin MIT-lisenssillä ja Serverless Frameworkia kehittää sen alkuperäinen kehittäjä Austen Collins ja Serverless Inc -yritys. [22]

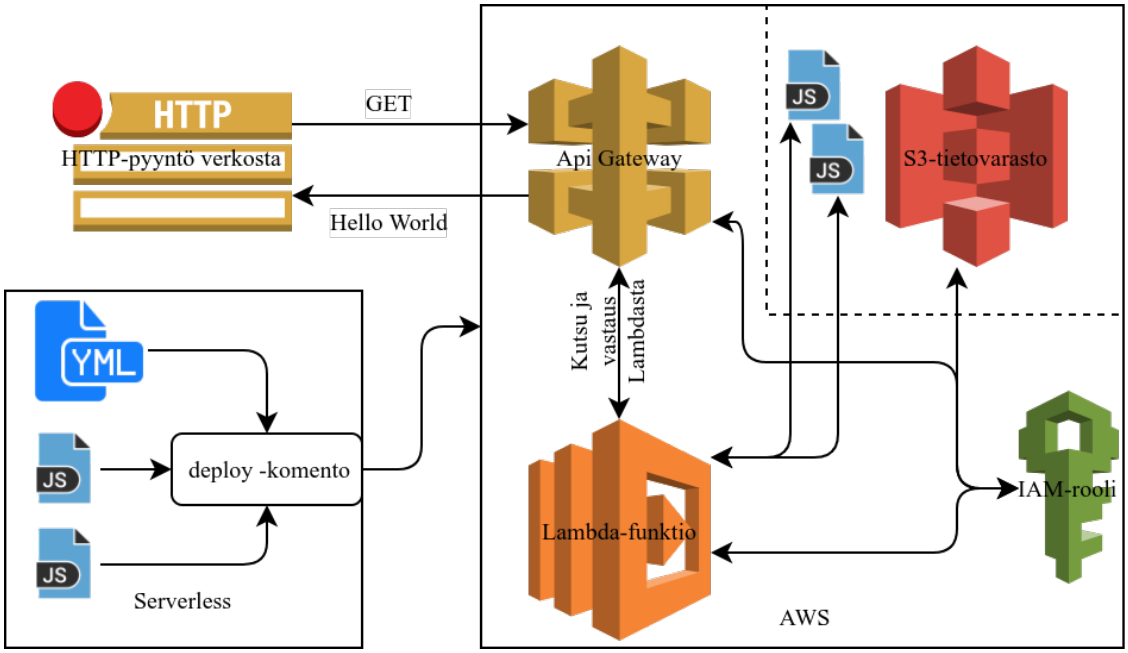
Serverless Framwork pyrkii yhtenäistämään vastaavien komentojen tekemistä eri alustoilla. Käyttäjä määrittelee *serverless.yml* tiedostoon millaisen infrastruktuurin haluaa julkaistavan tiedostossa määritellylle palveluntarjoajalle. Tällöin käyttäjän tulee vain opetella Serverless Frameworkin komennot ja syntaksi, eikä jokaisen palveluntarjoajan omia komentoja erikseen. Kuitenkin useissa tapauksissa, varsinkin monimutkaisempien infrastruktuurien rakentamisessa, ei voi vain suoraan vaihtaa palveluntarjoajaa toiseksi, vaan joitain pieniä yksityiskohtia täytyy myös muuttaa. Tälläisiä ovat muun muassa eri herätteitä antavien palveluiden nimet ja niiden uniikit ominaisuudet. Lisäksi käyttäjän täytyy määrittää kirjautumistiedot, joita sovellus käyttää kyseisessä palveluun. [22]

Kuitenkin esimerkiksi AWS Lambdan ja Api Gatewayn liittäminen toisiinsa on Serverless Frameworkin avulla huomattavasti yksinkertaisempaa, kuin esimerkiksi suoraan selainkäyttöliittymän kautta AWS konsolista. Kun kaikki vaiheet tehdään selaimen kautta manuaalisesti, täytyy ottaa huomioon miten Api Gateway, Lambda, S3-tietovarasto ja IAM-rooli liitetään oikein toisiinsa. Serverless Framework luo automaattisesti kaikki nämä ja liittää ne oikein toisiinsa. Esimerkkinä voidaan käyttää yleisesti teknologioiden esittelyyn käytettävää 'Hello World!' -sovellusta. Perustapauksessa voitaisiin haluta ajaa Lambda-funktio, joka palauttaa 'Hello World!'-tekstin, kun sitä kutsutaan Api Gatewayn tarjoaman HTTP-rajapinnan ylitse. Käytettäessä selaimesta AWS-konsolia, kehittäjä joutuu ensin luomaan S3-tietovaraston, jonne kehittäjä lataa lähdekoodit. Tämän jälkeen luodaan uusi Lambda-funktion, jolle määritetään käsittelijäfunktioksi S3-tietovarastossa sijaitsevan ZIP-pakkauksen mukana olevan funktio. Samalla luodaan uusi HTTP-rajapinta Api Gatewayn avulla ja liitetään näihin kaikkiin myös uusi IAM-rooli, jolla sallitaan palveluiden välinen kommunikaatio. Serverless Frameworkin avulla kaikki nämä luodaan automaattisesti listauksen 3.1 mukaisesti. Esimerkissä kaikkia palveluita ei ole pakko määritellä, koska Serverless Framework käyttää tarvittaessa oletusarvoja. Muun muassa S3-tietovarastoa ei ole pakko määritellä, mutta se on kuitenkin tarvittaessa mahdollista.

```
1  service: hello
2  provider:
3    name: aws
4    runtime: nodejs6.10
5  functions:
6    hello:
7      handler: handler.hello
8      events:
9        - http:
10           path: /
11           method: get
```

Listaus 3.1 *Serverless.yml konfiguraatiotiedosto 'Hello World!' -sovellukselle*

Kuvassa 3.6 on kuvattu millaisen AWS-infrastruktuurin Serverless Framework 'deploy'-komento tuottaa 'Hello World'-sovellukselle, sekä millaisia liityntöjä niillä on toisiinsa.



Kuva 3.6 *Serverless Frameworkin tuottama infrastuktuuri 'Hello World'-sovellukselle*

Kuvassa Serverless Frameworkin ajoympäristössä olevan Javasript-tiedostot on kopioitu S3-tietovarastoon ja AWS Lambda käyttää niissä olevaa käsittelijäfunktiota. IAM-rooli oikeuttaa eri palvelut toimimaan toistensa kanssa ja Api Gateway toimii HTTP-rajapintana verkosta tuleville GET-pyynnöille.

Lisäksi Serverless Frameworkin avulla on helppo rakentaa samanlainen, hyväksi todettu infrastuktuuri uudelleen, koska kaikki on määritelty koodina. Serverless Framework osaa myös päivittää ja alasajaa rakentamansa infrastuktuurin yhdellä komennolla. Alasajon jälkeen AWS-palveluja ei jää vahingossa turhaan käyntiin, vaan kaikki sammutetaan ja poistetaan. Näiden ominaisuuksien ansiosta Serverless Framework helpottaa huomattavasti normaaleissa kohtalaisen yksinkertaisissa tapauksissa serverless-mallin mukaisten tietojärjestelmien infrastruktuurin rakentamista.

On olemassa myös muita vastaavia palveluita, joilla voi määritellä koodin avulla järjestelmän infrastuktuurin, kuten esimerkiksi Amazonin oma tuote CloudFormation [23]. Sen ja muiden vastaavien palveluiden käyttöön ei tämän diplomityön puitteissa tutustuta. Serverless Framework keskittyy pääosin nimensä mukaisesti serverless-palveluiden ja niihin liittyvien osien rakentamiseen useilla eri palveluntarjoajilla. Sen sijaan CloudFormation keskittyy pelkästään Amazonin palveluvalikoimaan ja sillä voidaan rakentaa monipuolisempia kokonaisuuksia kuin Serverless Frameworkilla.

4. SERVERLESS-PALVELUNTARJOAJIEN TUOTTEIDEN OMINAISUUKSIEN VERTAILU

Tässä luvussa perehdytään lyhyesti markkinoilla oleviin FaaS-palveluntarjoajihin. Näkökulmaksi on valittu vertailu tässä diplomityössä käytettyyn AWS Lambda -palveluun. Lisäksi huomioidaan muut liitännäispalvelut, jotka tavanomaisesti ovat tarpeellisia FaaS-palvelun kanssa. Vertailussa käsitellään muun muassa hintatietoja ja palveluiden laajuutta.

4.1 Amazon

AWS Lambda on Amazonin tarjoama serverless-laskentayksikkö. AWS Lambdasta ja esimerkiksi sen käynnistävistä herätteistä on kerrottu tarkemmin Luvussa 3. Siinä mainittujen herätteiden lisäksi AWS Lambdat voivat saada herätteitä useista muista AWS-palveluista kuten Kinesis Streams, Simple Notification Service, Simple Email Service, Cognito, Cloudformation, CodeCommit, Alexa sekä Lex [11]. Erikseen korostettakoon Luvussa 3.3 käsitelty AWS Api Gateway palvelu, joka mahdollistaa lähes minkä tahansa muun toimittajan palvelun toimimisen herätteenä AWS Lambdalle HTTP-pyynnön avulla.

AWS Lambdan hinnoittelu lasketaan gigatavusekunti(GB-sekunti)-yksiköllä, jossa ajoaika otetaan huomioon 100 ms tarkkuudella. GB-sekunti tarkoittaa käytettyä sekuntia valitulla laskentayksikön muistin määrällä. Esimerkiksi 1 GB muistin laskentayksikköä ajettaessa yksi sekunti, on kulutettu yksi GB-sekunti. Amazon tarjoaa miljoona kyselyä ja 400 000 GB-sekuntia kuukaudessa ilmaiseksi, jonka jälkeen Amazon laskuttaa $0,20 per kysely ja $0,00001667 per käytetty GB-sekunti. Lambdalle on valittavissa useita eri muistin määriä. Kustannuksia sekä yksittäistä suoritusaikaa on vaikeaa laskea etukäteen, koska valittu muistin määrä määrittää myös käytettävän laskentatehon Lambdalle. Käytännössä vaikka sovellus käyttäisi vain vähän muistia, voi joissain tapauksissa olla järkevää valita siitä huolimatta suurempi muistimäärä ja samalla myös tehokkaampi laskentayksikkö. Siten suoritusaika jää pienemmäksi. [14]

AWS Lambda -funktio skaalautuu helposti ilman suuria kustannuksia. Suorituspiikkien

kohdalla kustannukset kasvavat suhteellisesti enemmän, mutta useampien funktioiden herättäminen tapahtuu automaattisesti. Hiljaisina aikoina suorituskertoja eikä siten kustannuksiakaan tule.

4.2 Microsoft

Microsoft Azure Functions on vastaava palvelu kuin Amazonin AWS Lambdat. Functions -palvelua käyttämällä kehittäjän ei tarvitse välittää koko sovelluksesta tai infrastruktuurista jolla yksittäistä funktiota ajetaan. Azure Functions tukee natiivisti C#, F#, Node.js, Python ja PHP-ohjelmointikieliä. Azure functions -palveluun voi liittää OAuth-autentikaatiopalveluita muilta ulkoisilta palveluntarjoajilta, kuten Google, Facebook, Twitter tai Azure Active Directory. [24]

Kuten AWS Lambdoissakin, Azure Functions -palvelussa ajettavan funktion voi käynnistää useilla eri herätteillä. Herätteinä voivat toimia esimerkiksi HTTP-pyyntö, ajastus, muutos Githubissa olevassa versiohallinnassa, muutokset Azuren pilvipalveluissa oleviin resusseihin, kuten datan lisääminen tietokantaan tai erilaisiin viestijonoihin tulleet viestit. [24]

Azure Functions hinnoittelu pohjautuu AWS Lambdoja vastaavasti suorituskertoihin ja resurssien käyttöön. Suorituskerroista veloitetaan $0,20 miljoona suoritusta kohden ja lisäksi resursseista veloitetaan käytettyjen GB-sekuntien perusteella $0,000016 per GB-sekunti. Azure Function -funktion hinnoittelussa muistimäärä perustuu keskimäärin käytettyyn muistimäärään. Tämän kulutukseen perustuvan hinnoittelun sijaan on mahdollista valita myös hinnoittelu, joka sisältyy muiden palveluiden kanssa niputettuun hinnoitteluun. Sitä ei käsitellä tässä yhteydessä, sillä sen vertaaminen muihin palveluntarjoajiin on vaikeaa. [25]

Kuten AWS Lambda, Azure Functions -funktiot ovat myös automaattisesti skaalautuva tuote, sillä se käynnistetään jokaiselle herätteelle erikseen ja kustannukset syntyvät vain suorituskerroista.

4.3 Google

Google Cloud Functions -palvelu on Googlen vastine AWS:n ja Azuren vastaaville palveluille. AWS ja Azure -palveluista poiketen, Googlen Cloud Functions -palveluun voi tehdä funktioita vain Node.js -kielellä [26]. Herätteinä Cloud Functions -funktiolle voivat toimia HTTP-pyynnöt, Googlen Cloud Storage tietovarasto, Google Cloud Sub-/Pub tapahtumavirrat, Googlen Firebase palvelun tietokanta-, varastointi-, analytiikka- ja autentikointitapahtumat sekä Google Stackdrive Logging -palvelun tapahtumat [27].

Googlen hinnoittelu pohjautuu käytetyn muistin, suoritintehon ja -ajan suhteeseen kahdella eri yksiköllä mitattuna, jotka ovat GB-sekunti ja GHz-sekunti. GB-sekunti maksaa $0,0000025 ja GHz-sekunti maksaa $0,0000100. Näistä kertyvät summat lasketaan yhteen valituilla teho- ja muistimäärillä sekä kulutetulla ajalla. Edellä mainitun lisäksi Google veloittaa yli 2 miljoonan kerran menevistä funktion suorituksista $0,40 per kaksi miljoonaa suoritusta. Google Cloud Functions -funktiolle voi itse valita haluamansa suoritintehon ja hyödynnettävän muistin määrän, joihin hinnoittelu perustuu. [28]

4.4 Havainnot

Tämän luvun havaintojen perusteella voidaan todeta, että jokainen näistä palveluista on edullinen suorituskertaa kohden, helposti skaalautuva suurempiin määriin suorituksia, resurssien määrän suhteen muunneltavia ja liitettävissä useisiin erilaisiin herätteisiin. AWS Lambdat sekä Azure Functions -palvelut ovat yhteensopivia useiden eri ohjelmointikielien kanssa, Google Functions on rajoitettu vain Node.js -ympäristöön, eli Javascript -ohjelmontikieleen. Kaikilla palveluntarjoajilla on jonkinlainen ilmainen osuus hinnoittelumallissa, mikäli funktion käyttö on hyvin suppeaa.

Merkittävä huomio lienee, että jokainen palvelu pystyy käyttämään herätteenä mielivaltaisia HTTP-pyyntöjä, ja siten ne ovat kaikki helposti liitettävissä moniin nykyisiin olemassaoleviin tietojärjestelmiin kohtalaisen helposti, vaikka muut osat järjestelmästä eivät pohjautuisi mitenkään näiden palveluntarjoajien tuotteisiin.

5. AFTERWORK-ALERT-SOVELLUS

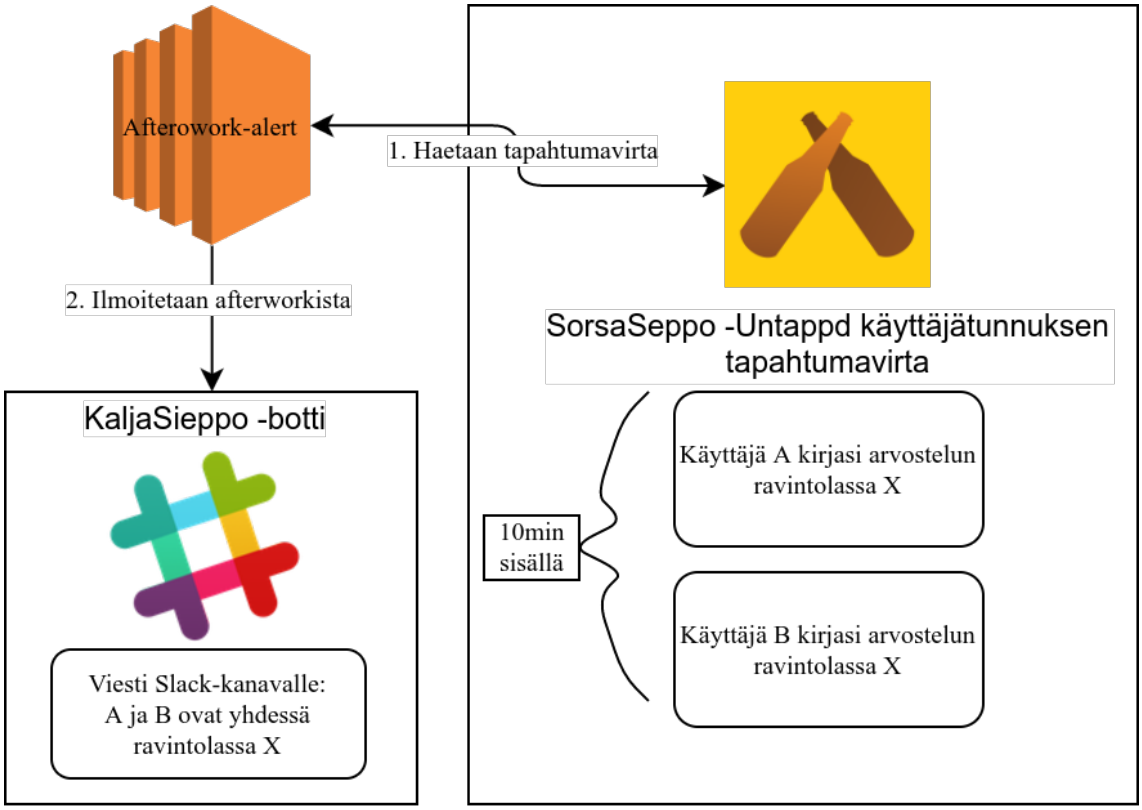
Tässä luvussa käsitellään Afterwork-alert-sovellusta, joka yhdistää Untappd ja Slack -palvelut. Tässä luvussa esitellään sovelluksen alkuperäinen rakenne sekä toimintalogiikka ja luvussa 6 esitellään uusi toteutus.

5.1 Sovelluksen tarkoitus ja logiikka

Sovellus on kehitetty ohjelmistoyritys Goforen työntekijöiden yhteisten illanviettojen tueksi. Sovelluksen tarkoitus on ilmoittaa automaattisesti kuvan 5.1 mukaisesti kun kaksi tai useampi työkavereista on viettämässä "afterworkia" samassa ravintolassa. Henkilöiden tunnistaminen tapahtuu Untappd-sovelluksen käytön kautta. Kuvassa 5.2 on kuvattu sovelluksen perustoiminnallisuus yleisellä tasolla. Henkilöt arvostelevat juomansa Untappd-sovelluksella, jolloin heidän Untappd-kaverinsa voivat nähdä arvostelut ja niiden sijaintitiedot omassa Untappdin tapahtumavirrassaan. Tälläistä tilannetta, jossa useampia työkavereita on yhtä aikaa samassa paikassa, kutsutaan afterworkiksi. Afterwork-alert toimii integraationa Untappdin ja Slackin välillä. Lisäksi Afterwork-alertille voi myös antaa joitain komentoja Slack viestien avulla. Sovellusta varten on luotu Untappd-palveluun käyttäjätunnus Goforen virtuaalisen maskotin SorsaSepon nimellä.

Kuva 5.1 *Slack-botin ilmoitus seurueesta sijainnissa Gofore Oy*

Afterwork-alert-sovellusta käyttämällä Goforelaiset voivat liittyä SorsaSepon kavereiksi Untappd-palvelussa. Afterwork-alert seuraa kavereidensa toimintaa Untappd-palvelusta ja ilmoittaa määrätyistä asioista Slackiin kaljasieppo -nimisen bottitunnuksen kautta. SorsaSeppo ei tee Untappd-palvelussa itse kirjauksia. Tunnuksen avulla ainoastaan seurataan tapahtumavirtaa SorsaSepon kavereiden toiminnoista ja Slackissa annettujen komentojen mukaisesti luodaan tai hyväksytään kaveripyyntöjä Untappd:ssa. Sovelluksen toimintaa varten on myös luotu Slackiin botti-käyttäjä, jonka kautta sovellus voi seurata komentoja Slackista sekä tarvittaessa lähettää viestejä kanaville.



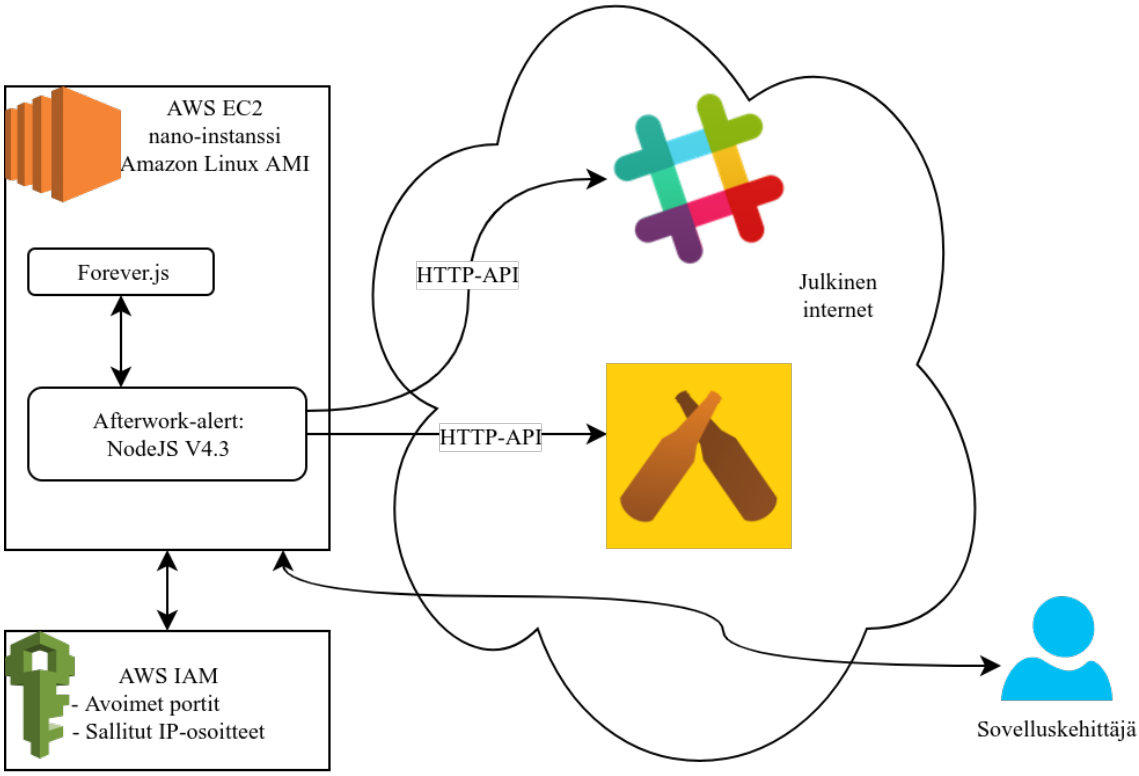
Kuva 5.2 *Afterwork-alert-sovelluksen perustoiminallisuus*

Afterwork-alert on konfiguroitu tutkimaan 10 minuutin välein onko seurue SorsaSepon Untappd-kavereita kirjannut juomia samassa sijainnissa riittävän samanaikaisesti. Seurueeksi tunnistetaan joukko Untappd-käyttäjiä, jotka ovat kirjanneet 30 minuutin sisällä samassa sijainnissa juoman arvostelun. Mukaan lasketaan myös sellaiset kirjaukset, jotka ovat luodun seurueen viimeisestä kirjauksesta alle 10 minuutin sisällä tehtyjä. Lisäksi Afterwork-alert tunnistaa, jos henkilön tekemä kirjaus on ollut mukana aiemmassa ilmoituksessa Slackiin, eikä käytä samaa kirjausta enää seuraavan seurueen muodostamiseen uudelleen.

5.2 Sovelluksen rakenne ja toiminta

Afterwork-alertin sovellusympäristönä toimii Amazon EC2-instanssi, johon on valittu käyttöjärjestelmän näköislevyksi Amazonin oma Amazon Linux AMI (Amazon Machine Image), joka sisältää tavallisiin ohjelmistokehitystarpeisiin valmiin ympäristön. Palvelimen kooksi valittiin pienin mahdollinen nano-koon palvelin. Palvelimen konfigurointiin liittyi myös turvallisuusryhmän (security group) luominen Amazonin palveluihin IAM-palvelussa, ja sen liittäminen tälle EC2-instanssille. Turvallisuusryhmä määriteltiin siten että vain vain tietyistä verkko-osoitteista ja vain tietyillä SSH-avaimilla voi ottaa yhteyttä sen alaisiin resursseihin. EC2-instanssin käyttöjärjestelmää ylläpidetään itse. Kuvassa

5.3 on havainnollistettu mitä eri osia sovellukseen liittyy.



Kuva 5.3 *Afterwork-alert-sovelluksen infrastruktuuri*

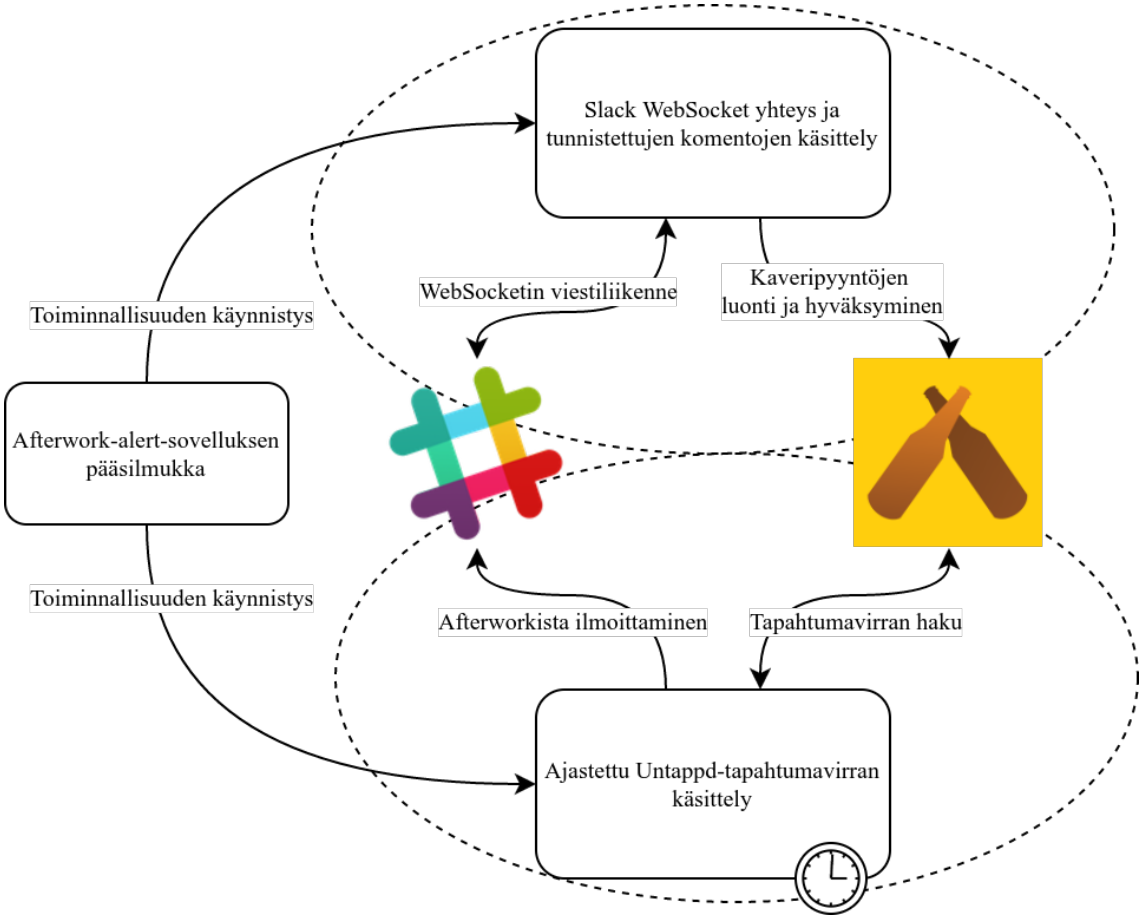
Sovellus on tehty Javascript -ohjelmointikielellä hyödyntäen joitain ECMAScript 6 ominaisuuksia ja sitä ajetaan Node.js -ajoympäristössä Forever.js -työkalun avulla. Forever.js huolehtii Node.js sovelluksen uudelleenkäynnistymisestä virheellisen sammumisen jälkeen. Sovellus käyttää Untappd-sovelluksen API:a node-untappd-kirjaston [29] avulla ja Slackin rajapintaa slack-node-kirjaston [30] kautta. Muita hyödynnettyjä Node.js-kirjastoja ovat *lodash* [31] erilaisten tietovirtojen muovaamiseksi, *moment* [32] ajan käsittelyn helpottamiseksi, *https* salattujen yhteyksien takaamiseksi sekä *ws* [33] Slackin rajanpinnan WebSocketin käyttöä varten. Listauksessa on myös esitetty paikallisessa ympäristössä sijaitsevan config.json asetustiedoston liittäminen sovellukseen.

Sovelluksen ajon alussa luodaan Untappd-rajapintaa käyttävä asiakasohjelma Listauksen 5.1 rivien 1-5 mukaisesti. Slackin asiakasohjelma määritellään rivien 6-7 mukaisesti. Parametreina annetut *debug*, *clientId*, *clientSecret*, *accessToken* ja *slackApiToken* ovat määritelty config.json asetustiedostossa.

```
1 // Create Untappd Client
2 var untappd = new UntappdClient(debug);
3 untappd.setClientId(clientId);
4 untappd.setClientSecret(clientSecret);
5 untappd.setAccessToken(accessToken);
6 // Create Slack Client
7 var slack = new Slack(slackApiToken);
```

Listaus 5.1 *Untappd ja Slack-rajapintojen käyttöönotto*

Sovellus koostuu kahdesta loogisesti toisistaan eroavasta toiminnallisuudesta. Ensimmäisenä on 10 minuutin välein ajastetusti ajettava Untappd-rajapinnasta haettavan datan käsittely ja jäsennys seurueiksi sekä näistä Slackiin ilmoittaminen. Toinen selkeä osio sovelluksessa on kaveripyyntöjen hyväksyminen ja tekeminen Slackissa annettujen komentojen mukaisesti. Molemmat osiot käyttävät samoja ulkoisia kirjastoja ja palveluita, mutta ne pystyvät toimimaan myös itsenäisesti ilman toisiaan. Näiden toiminnallisuuksien irrallisuus on havainnollistettu Kuvassa 5.4.



Kuva 5.4 *Sovelluksen toiminnallisuuksien erottuminen*

Listauksessa 5.2 esitetään kuinka ensimmäistä osaa varten luodaan myöhemmin ajastettuna kutsuttava funktio, joka hakee Untappd-tapahtumavirran 25 uusinta tapahtumaa, tun-

nistaa afterwork-tilanteet ja sen perusteella luo ja lähettää Slackiin viestin.

```
// Helper for interval
function timer() {
  getUntappdFeed()
    .then(parseAfterworkers)
    .then(buildPayloads)
    .then((resolve, reject) => {
      resolve.map((payload) => {
        slack.api("chat.postMessage", payload, function (err, response)
    {
          log("SLACK response: ", response);
        })
        log("SLACK resolve: ", payload);
      });
    })
    .catch((reason) => {
      log("ERROR reason: ", reason);
    });
}
```

Listaus 5.2 *Tapahtumavirran jäsentäminen*

Toista toiminnallisuutta, eli komentojen vastaanottamista varten luodaan Listauksen 5.3 mukaisesti WebSocket aiemmin käyttöönotetun Slack-asiakasohjelman avulla ja kuunnellaan WebSocketin kautta tulevia viestejä.

```
// Helper for starting to follow slack
function followSlack() {
  slack.api('rtm.start', function(err, response) {
    slack.api('auth.test', function(err, res) {
      listenWebSocket(response.url, res.user_id);
    });
  });
}
// WebSocket listening for commands
function listenWebSocket(url, user_id) {
  var ws = new WebSocket(url);
  ws.on('open', function(message) {
    // Handling code here
  }
  ws.on('message', function(message) {
    // Handling code here
  }
}
```

Listaus 5.3 *WebSocketin seuraaminen ja ajastuksen käynnistäminen*

Sovelluksen lopussa kutsutaan aiemmin Listauksissa 5.2 ja 5.3 määriteltyjä funktioita Listauksen 5.4 mukaisesti. Rivillä 2 kutsutaan ajastetusti 10 minuutin välein Untappd-tapahtumavirran käsittelyä ja rivillä 3 kutsutaan WebSocket yhteyden käynnistävää funktioita.

```
1 // Actual calls for starting different parts of application
2 setInterval(timer, loopingTime * 1000 * 60);
3 followSlack();
```

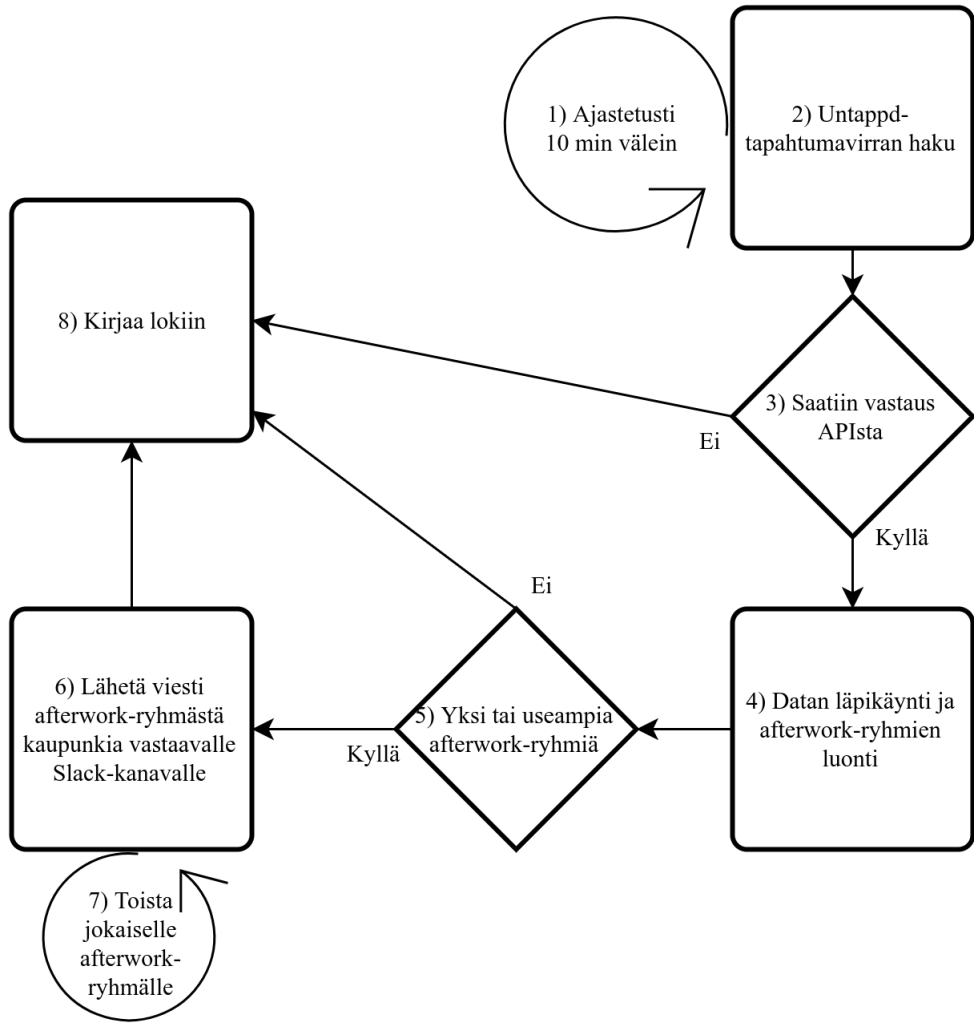
Listaus 5.4 *Websocketin käynnistäminen sekä ajastuksen luominen tapahtumavirran jäsentämiselle*

5.2.1 Untappd-tapahtumavirran jäsentäminen seurueiksi

Kuvassa 5.5 on kaavio Untappd API:sta haettujen tapahtumien jäsentämisestä. Sovellus hakee ajastetusti kuvan kohtien 1) ja 2) mukaisesti tapahtumavirran 10 minuutin välein Untappd-API:sta. API:n vastauksesta tallennetaan oleelliset tiedot jatkokäsittelyä varten. Tälläisiä tietoja ovat *checkin_id*, *created_at*, *venue.venue_id*, *venue.venue_name*, *venue.location.venue_city*, *user.uid*, *user.first_name*, *user.last_name*. Kuvan kohdassa 4) näiden tietojen avulla sovellus pystyy jäsentämään, mitkä kirjaukset liittyvät samaan seurueeseen. Jos seurueita on yksi tai useampi, lähetetään seurueen kaupungin mukaiselle kanavalle ilmoitus Slackiin. Kaupungit määräytyvät Goforen toimipaikkojen sijaintien mukaisesti. Tämä toistetaan jokaiselle seurueelle erikseen, eli viestejä voidaan lähettää usealle eri kanavalle, jos seurueita on eri kaupungeissa yhtä aikaa. Kaikista toiminnoista kirjoitetaan konsoliin lokituloste, joka tässä tapauksessa ohjautuu forever.js sovelluksen kirjoittamiin lokeihin.

Kuvan 5.5 kohdassa 4) kerätty data käsitellään hyödyntäen Moment.js ja Lodash-kirjastoja. Moment.js kirjaston avulla ajan käsittely on helpottuu ja Lodash on datan manipulointiin soveltuva työkalu. Lodash-kirjaston avulla putkitetaan useita toimenpiteitä:

1. järjestetään kirjaukset aikajärjestykseen
2. suodatetaan pois liian vanhat kirjaukset config.js mukaisesti
3. suodatetaan pois käytetyt kirjaukset, jotka on kirjattu ajonaikaisesti sovelluksen muistiin
4. suodatetaan pois kirjaukset ilman sijaintia
5. ryhmitellään kirjaukset sijainnin mukaan



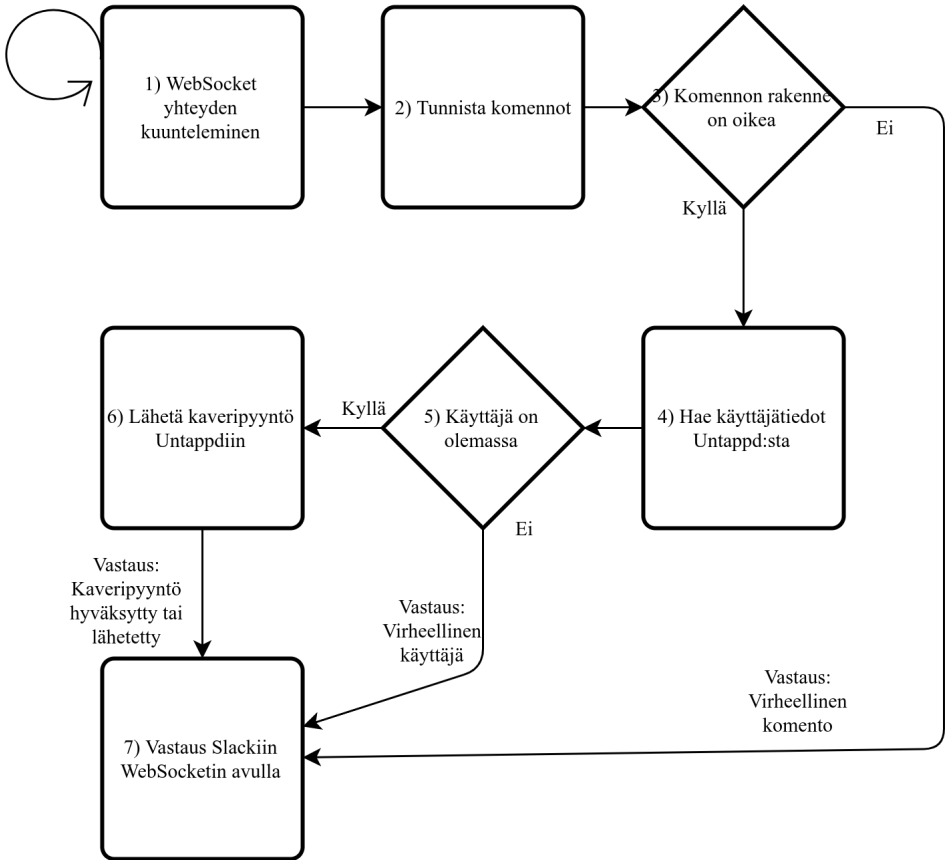
Kuva 5.5 *Afterwork-alert-sovelluksen Untappd API:sta haettujen tapahtumien jäsentämisen kaavio*

6. käsitellään ryhmät sisäisesti vanhimmasta uusimpaan muodostamalla seurue 30 minuutin sisällä toisistaan tehdyistä kirjauksista tai alle 10 minuutin sisällä edellisestä seurueeseen kuuluneesta kirjauksesta
7. suodatetaan ryhmät joissa on alle kaksi kirjausta

Esitetyn Kuvan 5.5 avulla sovellus pystyy muodostamaan uudet seurueet, joista se ilmoittaa 10 minuutin välein Slackiin kaupunkikohtaisille kanaville.

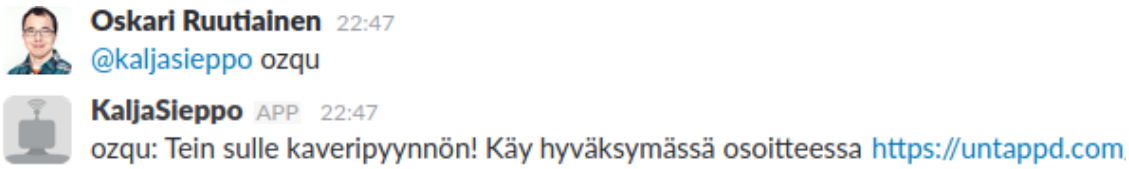
5.2.2 Kaveripyynnön tekeminen tai hyväksyminen Slack-viestien perusteella

Kuvassa 5.6 on kaavio kaveripyyntöjen tekemisestä Slack-viestin perusteella. Slack-viesti on Kuvan 5.7 mukainen.



Kuva 5.6 *Afterwork-alert-sovelluksen kaveripyynnön kaavio*

Sovellus ottaa yhteyden Slackiin WebSocketin avulla käynnistyksen yhteydessä. Tämän jälkeen kuvan kohdan 1) mukaisesti sovellus seuraa viestejä niillä kanavilla, joille botti-käyttäjä on liitetty Slackissa. Sovellus tunnistaa viesteistä ne, jotka alkavat botin nimimerkillä. Slackissa on sisäänrakennettu ilmoitusmekanismi, joka toimii lisäämällä ”@”-merkin käyttäjän nimimerkin eteen. Slackin rajapinta palauttaa tälläiset nimimerkit ja kanavien nimet enkoodattuna. Sovellukseen on kehitetty tuki sekä dekoodatun nimimerkin tunnistamiselle, että nimimerkin tunnistaminen normaalista merkkijonosta.



Kuva 5.7 *esimerkki käyttäjän tekemästä kaveripyyntökomennosta Slackissa*

Kuvan 5.6 kohdan 3) mukaisesti virheellisestä komennosta lähetetään Slackiin vastaus, jossa ohjeistetaan millainen komennon täytyy olla. Oikeanlaisen komennon mukaan sovellus pyytää Untappd-API:sta kohteena olevan käyttäjän tiedot kohdassa 4). Jos käyttäjää ei ole olemassa, sovellus ilmoittaa Slackiin virheellisestä käyttäjästä. Jos käyttäjä on olemassa, sovellus tekee kaveripyynnön Untappd-API:n avulla ja ilmoittaa Slackin API:n

kautta komennon antajalle. Tässä tilanteessa luotiin joko uusi kaveripyyntö, hyväksyttiin käyttäjän SorsaSepolle tekemä kaveripyyntö tai havaittiin että kaveripyyntö on tehty aiemmin, mutta käyttäjä ei vielä ole hyväksynyt sitä.

5.3 Käyttökokemukset

Ennen serverless-muunnosta sovellus oli käytössä joitain kuukausia Goforen työntekijöiden kesken. Sovelluksen toiminnallisuudet olivat kunnossa, mutta pidempiaikaisessa yhtäjaksoisessa käytössä havaittiin joitain ongelmia.

Sovelluksessa ilmeni esimerkiksi satunnaisia virheellisiä sammumisia. Syyksi todettiin sovelluksen WebSocket-yhteys Slackiin, joka vaati uudelleenyhdistämistä eri verkko-osoitteella säännöllisin välein. Uusi osoite saatiin olemassa olevan WebSocket-yhteyden yli. Mikäli tätä uudelleenyhdistämistä ei tehty oikealla tavalla, sovellus kaatui ja käynnistyi uudelleen, yrittäen luoda yhteyden täysin alusta saakka. Tämän olisi teoriassa pitänyt toimia, koska silloin sovellus pyytää täysin uuden osoitteen johon yhdistää. Todennäköisesti Slack on sillä hetkellä sellaisessa tilassa, että se odottaa yhdistämistä juuri antamansa uudelleenyhdistämisosoitteen kautta. Tästä syystä uudelleenkäynnistyminen ei toiminut konsistentisti ja sovellus jäi ajoittaisesti uudelleenkäynnistymissilmukkaan. Sovellus oli myös asetettu lähettämään Slackiin ilmoitus käynnistymisestään. Tämä johti siihen että tällaisen silmukkaan jäämisen aikana sovellus lähetti muutaman sekunnin välein kehittäjien Slack-kanavalle viestin, kunnes sovellus suljettiin ja käynnistettiin manuaalisesti uudelleen. Lisäksi erinäisten palomuurisääntöjen takia kuka tahansa ei voinut kirjautua käytössä olevalle EC2-instanssille. Tällöin kyseinen WebSocket-ongelma saattoi kestää muutamia tunteja, ennen kuin henkilö, jolla on tarvittavat oikeudet, pääsi sallitusta verkko-osoitteesta käsiksi virtuaalikoneelle tekemään uudelleenkäynnistyksen sovellukselle.

Toinen käytönaikainen havainto oli huonosti valittu sovelluksen tilanhallinta. Sovelluksen tilaa säilytettiin ajonaikaisessa muistissa, joten tila tyhjentyi aina sovelluksen sammumisen yhteydessä. Tila olisi ollut parempi säilyttää tietokannassa, jolloin uudelleenkäynnistäminen ei olisi vaikuttanut sovelluksen toimintaan. Lisäksi tietokannan avulla serverless-muunnoksenkin jälkeen uusi sovellus olisi voinut hyödyntää olemassa olevaa tilanhallintaa ilman erillisiä muutoksia tai kompromisseja sovelluslogiikan toteutuksessa.

Sovellus sai käytön aikana kuitenkin paljon kiitosta työyhteisössä, ja se nähtiin mielenkiintoisena harrasteprojektina, keskustelunherättäjänä teknologioista ja yhteiseen illanviettoon kannustavana. Tästä syystä nähtiin tarpeelliseksi myös kehittää sovellusta teknisesti paremmalle pohjalle.

6. SOVELLUKSEN MUUTTAMINEN SERVERLESS-PALVELUITA HYÖDYNTÄVÄKSI

Tässä luvussa käsitellään Luvussa5 kuvatun Afterwork-alert-sovelluksen muuttaminen serverless-palveluita hyödyntäväksi. Goforen työyhteisössä on paljon kokemusta Amazonin palveluista ja niiden kokemusten pohjalta tämänkin projektin kohdalla päädyttiin Amazonin palveluihin. Serverless-palveluista muutosprojektissa hyödynnettiin Amazon Web Services Lambdaa ja siihen liittyviä muita Amazon Web Services -tuotteita. Lisäksi esitetään kuinka uusi Afterwork-alert ja sen tarvitsema AWS-infrastruktuuri julkaistaan hyödyntäen Serverless Framework -työkalua Amazonin pilveen.

Sovelluksen muuttamisessa Amazon Web Services Lambdojen avulla ajettavaksi on useita erilaisia ratkaistavia haasteita. Sovelluksen tekniset ratkaisut täytyy päivittää soveltumaan AWS Lambdojen tarjoamalle ajoympäristölle. Ajoympäristö täytyy mahdollisesti konfiguroida eri tavalla kuin perinteisemmällä EC2-instanssilla Luvussa 5.1 ja mahdollisesti tehdä muutokset sovelluslogiikkaan AWS Lambdan tilattoman toiminnan takia. Edellä mainitut haasteet pystytään ratkaisemaan useilla eri tavoilla. Mahdollisia ratkaisutapoja vertaillaan tämän esimerkkiprojektin osalta tässä luvussa.

6.1 Ohjelmakoodin muutokset

Sovelluksen suurimmat käytännön haasteet koettiin uusien teknologioiden käytössä sekä sovelluksen algoritmien kanssa. AWS Lambda -konversion kehitysaikana AWS Lambda tuki Node.js V4.3 ajoympäristöä, joka ei tue Javascriptin uusinta EC6 standardia [34]. Osa käytetyistä ohjelmointia helpottavista rakenteista täytyi kirjoittaa uudelleen vanhemman standardin mukaisesti. Sovelluksessa ei kuitenkaan ole käytetty mitään erityisen korkean tason abstraktioita tai muita sellaisia ominaisuuksia, jotka varsinaisesti olisivat vaatineet uuden standardin käyttöä. EC6:n käyttö helpotti ohjelmointityötä ja sen avulla koodista tuli selkeämpää. Listaus 6.1 antaa esimerkkinä nuolifunktion poistamisen koodista. Nuolifunktion *this* avainsanan sitomista ei oltu hyödynnetty mitenkään, joten muutos oli hyvin triviaali syntaksin muuttaminen [35].

```
1  // EcmaScript6 nuolifunktioiden kanssa
2  afterwork = _.chain(feed)
3        .sortBy((checkin) => {
4          return moment(checkin.time, timeFormat);
5        })
6  // ...
7
8  // Ilman nuolifunktioita
9  var afterwork = _.chain(feed)
10       .sortBy(function(checkin) {
11         return moment(checkin.time, timeFormat);
12       })
13 // ...
```

Listaus 6.1 *Javascriptin nuolifunktioiden muutokset*

AWS Lambda tukee tällä hetkellä Node.js v6.10 -ajoympäristöä [11]. Uusin versio Node.js -ympäristöstä on tällä hetkellä v8.0.0 [36], joten uusien ominaisuuksien tarjoamat mahdollisuudet voivat helposti jäädä saavuttamatta, koska uusin versio ei ole AWS Lambdalla käytettävissä.

6.2 Sovelluslogiikan muutokset

Sovellukseen tukeutuminen ajonaikaisesti muistiin tallennettuun tietoon täytyi myös muuttaa AWS Lambdojen käyttöönotossa. Kun sovellus muodostaa Untappd-tapahtumavirrasta seurueen, sovellus tallentaa seurueeseen kuuluneiden kirjausten tunnisteet eikä se käytä näitä kirjauksia seuraavilla kerroilla tarkastellessaan tapahtumavirtaa. AWS Lambda on tilaton ja ajonaikaisesti sovelluksen muistiin tallennettu tieto ei ole käytettävissä seuraavalla ajokerralla. Sovellus vaatisi joko tietokannan, johon tallennetaan kirjausten tunnisteet, tai vaihtoehtoisesti seurueiden tunnistamisalgoritmia täytyy muokata soveltumaan käytettäväksi ilman tallennettuja arvoja. Yksi vaihtoehto olisi konfiguroida AWS DynamoDB -palveluun NoSQL-tietokanta, jonne tallennettaisiin sovelluksen käyttämät kirjaukset. DynamoDB kuitenkin tuottaa kustannuksia ja siten kasvattaa sovelluksen kerryttämiä kustannuksia. Kustannusten lisäksi täytyisi lisätä sovelluslogiikkaa tietokannan käsittelyä varten. Tietokannan käsittely myös lisäisi jonkin verran suoritusaikaa, joskaan ei merkittävissä määrin. Tietokannan käyttöönotto on myöskin jossain määrin serverless-mallin vastaista, vaikka siinäkin tapauksessa sovelluslogiikka toimii ilman sille varattua fyysista tai virtuaalista palvelinta.

Toinen vaihtoehto on muuttaa sovelluslogiikka sellaiseksi, ettei sovelluksen tarvitse tunnistaa aiemmin käytettyjä kirjauksia. Tämä on mahdollista toteuttaa usealla eri tavalla. Sovellus voi hakea Untappd-tapahtumavirran niin pitkältä ajalta, että se pystyy laskemaan

millaisia seurueita aiemmilla tarkastelukerroilla on muodostunut, ja jättää niiden kirjaukset huomioimatta seuraavan seurueen muodostamisessa. Tämä ratkaisu kuitenkin lisäisi suoritusaikaa, koska laskentaa täytyisi tehdä huomattavasti enemmän. Lisäksi Untappd-API palauttaa vain 25 kirjausta kerrallaan rajapinnasta, joten verkon yli tehtäviä kyselyjä tulisi huomattavasti enemmän, joka jälleen lisää suoritusaikaa.

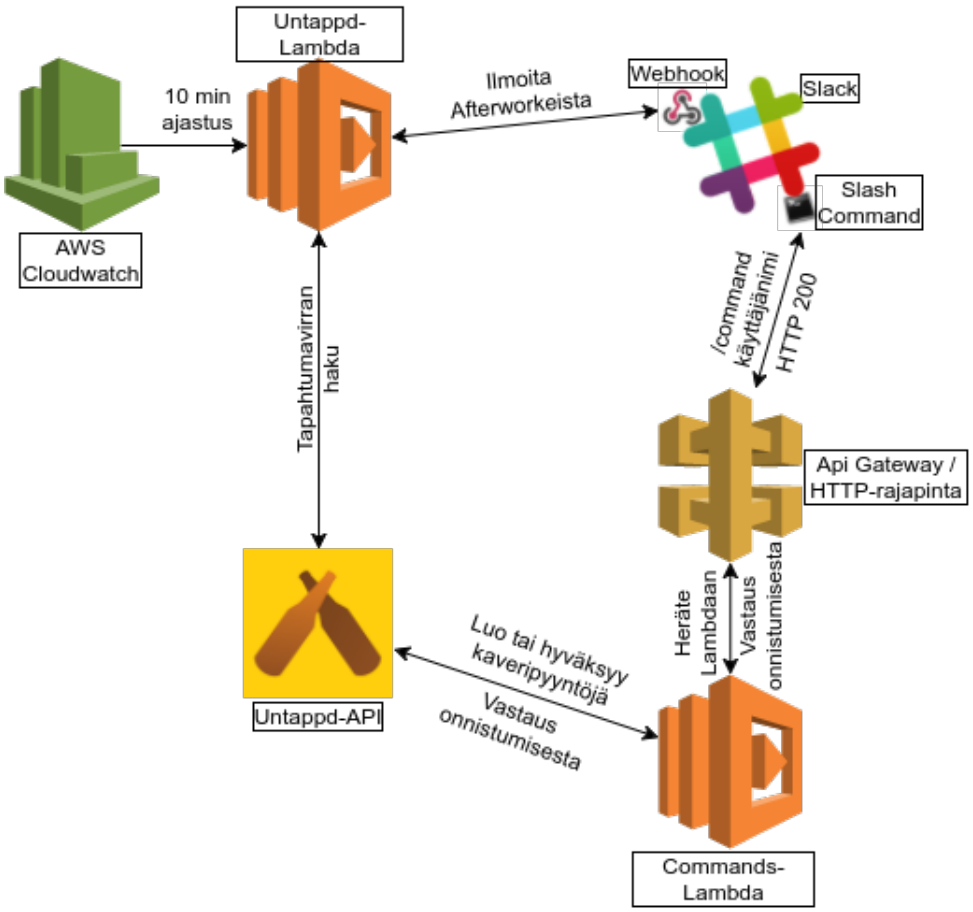
Käytännössä päädyttiin yksinkertaistamaan tapahtumavirran jäsennystä siten, että sovellus huomioi aina kaikki tapahtumavirrasta haetut kirjaukset, välittämättä siitä onko jotain yksittäistä kirjausta hyödynnetty edellisellä tapahtumavirran käsittelykierroksella. Untappd palauttaa yhdellä pyynnöllä tuoreimmat 25 tapahtumaa tapahtumavirrasta. AWS Lambdalla ajettava muutettu sovellus herätetään AWS Cloudwatchin herätteellä 10 minuutin välein ja sovellus tutkii edellisen 20 minuutin ajalta Untappd-tapahtumavirtaa. Tapahtumavirran jäsennystä Lodash-kirjaston avulla muutettiin siten, että 20 minuutin aikajaksolla täytyy olla vähintään kaksi eri käyttäjän kirjausta samassa sijainnissa, joista vähintään toinen on edellisen 10 minuutin ajalta. Tämän muutoksen takia yksittäistä kirjausta saatetaan käyttää kahdesti eri seurueessa, jos kirjauksia tulee jokaisella 10 minuutin aikajaksolla samasta sijainnista.

6.3 Ajoympäristön ja Slack-konfiguraatioiden muuttaminen

Alkuperäisessä sovelluksessa seurattiin Slackissa annettuja komentoja WebSocketin yli. Muutoksen yhteydessä WebSocket jätettiin täysin pois ja konfiguroitiin Slack seuraamaan itsenäisesti annettuja komentoja sen sisäänrakennetulla Slash Commands -ominaisuudella [37]. Slash Commands -ominaisuuden avulla Slack lähettää konfiguroituun verkko-osoitteeseen HTTP-pyynnön, joka sisältää tarvittavat tiedot Afterwork-alertin toimintoja varten. AWS Lambda ei ole suoraan verkon yli saatavilla, vaan kaikki sen hyväksymät herätteet tulevat muista AWS palveluista. Viestin vastaanottamista varten AWS Lambda tarvitsee seurakseen AWS Api Gateway -palvelun, joka määrittää HTTP-protokollan avulla kutsuttavan rajapinnan, joka sitten ohjaa kyselyt edelleen sisältöineen AWS Lambdalle. Kuvassa 6.1 on määritelty mitä eri osia uudessa sovelluksessa on ja millaisia herätteitä ja viestejä niiden välillä liikkuu.

Koska Lambdat käynnistyvät vain saatuaan herätteen, pystyttiin sovelluksesta poistamaan kokonaan muun muassa forever.js-kirjasto. Forever.js huolehti alkuperäisen sovelluksen käynnistymisestä automaattisesti uudelleen virheellisen sammumisen yhteydessä. Serverless-arkkitehtuurissa sovellus käynnistyy joka tapauksessa aina jokaisen ajon välissä uudelleen.

Sovellukselle määriteltiin yksinkertainen Api Gateway ja Slack-sovelluksen autentikointi rakennettiin AWS Lambdan sisään, hyödyntäen Slack Slash Commands -ominaisuuden



Kuva 6.1 *AWS Lamdboihin tukeutuvan sovelluksen arkkitehtuuri*

automaattisesti generoimaa avainta Listauksen 6.2 mukaisesti. Autentikoinnin voi rakentaa myös erilliseen AWS Lambdaan, joka on mahdollista liittää AWS Api Gateway -palveluun valtuutusfunktioksi [38]. Tällöin samaa valtuutusfunktioita voisi hyödyntää muissakin palveluissa [38]. Lisäksi Api Gateway pystyy tallentamaan välimuistiin samanlaisten kyselyjen oikeudet [38].

Valtuutusfunktion pitää palauttaa AWS Identity Manager -valtuutus Api Gatewaylle, joka tarkistaa onko kyseisellä valtuutuksella oikeus käyttää AWS Lambdaa [38]. Tässä tilanteessa kuitenkin nähtiin monimutkaisempien ratkaisujen olevan tarpeettomia tämän sovelluksen mittakaavassa, ja rakennettiin autentikointi AWS Lambdan sisään suoraan.

Kahden erillisen AWS Lambdan, AWS Api Gatewayn ja AWS Cloudwatchin konfigurointi on Amazonin oman selainkäyttöliittymän tai komentorivityökalun avulla kohtalaisen monimutkaista. Varsinkin kokonaisuuden hallinta ja asioiden toisiinsa liittäminen koettiin muutosprojektin yhteydessä vaikeaksi.

```
exports.handler = function(event, context, callback) {
  var body = _.chain(event.body)
  .split("&")
  .map(function(n) {
    return _.split(n, "=");
  })
  .fromPairs()
  .value();
  if (body.token !== slack_slash_token) {
    callback(null, {
      statusCode: 401
    });
  } else {
// ...
```

Listaus 6.2 *AWS Lambda - tapahtumakäsittelijän alkuun tehty autentikointi*

Selainkäyttöliittymää käyttäessä täytyy itse pitää tarkkaan kirjaa eri palveluiden resurssinimistä ja huolehtia navigoida useiden eri palveluiden välillä. Tähän helpotusta toi kuitenkin Serverless Framework -työkalun hyödyntäminen. Yksinkertaisen konfigurointitiedoston avulla pystyytän määrittämään suoraan millainen palvelukokonaisuus sovelluksille on tarkoitus luoda, jonka lisäksi työkalu luo ne automaattisesti. Serverless Framework tukee myös muita vastaavia serverless-palveluntarjoajia, mutta tämän projektin yhteydessä ei tutustuttu niihin. Listauksessa 6.3 on esitetty minkälainen konfiguraatiotiedosto riittää tämän projektin palvelukokonaisuuden luomiseksi. Serverless Framework abstrahoi yksinkertaisemmaksi AWS CloudFormation -palvelulla määriteltävän järjestelmän infrastruktuurin. Tämä konfiguraatio vastaa aiemmin Luvussa 3.5 Kuvassa 3.6 vasemmassa alareunassa olevaa *.yml* tiedostoa, jonka perusteella määritellään luotava infrastruktuuri.

```
service: afterwork-alert-feed

provider:
  name: aws
  profile: ${env:AFTWRK_PROFILE}
  runtime: nodejs6.10
  region: eu-west-1
  memory: 128

package:
  include:
    - untappdFeedParser.js
    - friendRequest.js
    - node_modules/**
  exclude:
    - ./**
```

```
17
18 functions:
19   untappd:
20     handler: untappdFeedParser.handler
21     environment:
22       CHANNEL_TAMPERE: '#afterwork-tampere'
23       CHANNEL_JYVASKYLA: '#afterwork-jkl'
24       CHANNEL_HELSINKI: '#afterwork-helsinki'
25       # If no channel for some city, use this
26       FALLBACK_CHANNEL: ${env:AFTWRK_FALLBACK_CHANNEL}
27       SLACK_WEBHOOK: ${env:AFTWRK_SLACK_WEBHOOK}
28       BOTNAME: ${env:AFTWRK_BOTNAME}
29       UNTAPPD_ACCESS_TOKEN: ${env:AFTWRK_UNTAPPD_ACCESS_TOKEN}
30       AFTERWORK_TIME_SEQUENCE: 20
31
32     events:
33       - schedule: rate(10 minutes)
34
35   # Handles commands
36   command:
37     handler: friendRequest.handler
38     environment:
39       SLACK_WEBHOOK: ${env:AFTWRK_SLACK_WEBHOOK}
40       SLACK_SLASH_TOKEN: ${env:AFTWRK_SLACK_SLASH_TOKEN}
41       UNTAPPD_ACCESS_TOKEN: ${env:AFTWRK_UNTAPPD_ACCESS_TOKEN}
42       BOTNAME: ${env:AFTWRK_BOTNAME}
43
44     events:
45       - http:
46           path: command
47           method: post
```

Listaus 6.3 *Serverless.yml konfiguraatiotiedosto*

Listauksen 6.3 alussa määritellään palvelukokonaisuuden nimi. *Provider* -lohkossa määritellään palveluntarjoaja, ajoympäristö ja käytettävä tunnus. *Package* -lohkossa määritellään mitä tiedostoja lähetetään palveluntarjoajan alustalle paketoituna. *Functions* -lohkossa määritellään kaksi eri AWS Lambda funktiota. *Untappd*-funktiolle määritellään käsittelijän sijainti, ympäristömuuttujat sovellusta varten sekä riveillä 36-37 herätteeksi ajastus. Serverless Framework luo listauksen määrittelystä AWS Cloudwatch -herätteen tälle AWS Lambdalle. Cloudwatch -herätteen voi määritellä myös monipuolisemmin, mutta tässä tapauksessa yksinkertainen ajastus on riittävä. Cloudwatch suorittaa sovelluksen serverless-arkkitehtuurissa samaa toiminnallisuutta, joka oli alkuperäisessä sovelluksessa toteutettu sovelluksen sisäisesti. Koodi esiteltiin Luvussa 5 Listauksessa 5.4.

Listauksessa 6.3 *command*-funktiolle määritellään samalla tavoin käsittelijä ja ympäristömuuttujat kuin edeltävälle *untappd*-funktiolle. Herätteeksi tälle määritellään HTTP-rajapinta riveillä 48-51. Serverless Framework luo tästä herätteestä AWS Api Gateway-palvelun muodostaman HTTP-rajapinnan. Kaiken tämän määrittely Serverless Frameworkin avulla on huomattavasti yksinkertaisempaa kuin kokonaisuuden rakentaminen selainkäyttöliittymän tai konsolityökalun avulla, koska eri AWS -palveluiden linkittäminen toisiinsa tapahtuu automaattisesti.

6.4 Havainnot

Muutosprojektin aikana havaittiin hyviä ja huonoja puolia AWS Lambdasta. Toisaalta täytyi ohjelmoida joitain kohtia uudelleen, sillä AWS Lambda ei tukenut riittävän uutta Javascript -standardia, mutta toisaalta muutokset olivat tässä tapauksessa lähinnä kosmeettisia. Yksin AWS Lambda -palvelu ei olisi riittänyt sovellukselle, joten sen tueksi täytyi konfiguroida myös AWS Api Gateway ja AWS Cloudwatch. Kuitenkin infrastuktuuri määriteltiin Serverless Framework -työkalun avulla. Kyseinen työkalu on helposti lähestyttävissä, vaikka sen käyttö tuottaakin hieman ylimääräistä opettelua. Varsinkin näin pienen projektin puitteissa Serverless Framework helpotti huomattavasti eri palveluiden infrastruktuurin määrittämistä ja palveluiden liittämistä toisiinsa. Infrastruktuurin määrittely ohjelmallisesti on vielä kohtalaisen uusi tapa luoda infrastruktuuri, ja on vaikeaa arvioida onko Serverless Framework juuri se työkalu, jota hyödynnetään jatkossa.

Muutosprojektissa havaittiin myös tarpeelliseksi pienten muutosten tekeminen sovelluslogiikkaan, kuitenkin säilyttäen loppukäyttäjän kannalta oleellinen toiminnallisuus. Kompromisseja oli järkevää tehdä, koska niiden avulla saavutettiin valmis sovellus pienemmällä työmäärällä ja ilman tietokantaa. Vaikka tietokanta olisi otettu palveluna, eikä se olisi maksanut paljon, sen käyttöönotto olisi ollut alkuperäisen tavoitteen vastainen. Yksi alkuperäisistä tavoitteista oli irtautua jatkuvasti käynnissä olevista palvelimista, jotka tuottavat kustannuksia, vaikka niitä ei käytettäisi aktiivisesti.

Serverless-sovelluksissa ei myöskään tarvitse huolehtia virhetilanteista palautumisesta samalla tavoin kuin perinteisien sovelluksien kohdalla. Virhe esimerkiksi kolmannen osapuolen kirjaston virheen tai verkkovirheen takia ei yleensä aiheudu kuin yksi virheellinene suoritus sovellukselle. Aiemman sovellusversion olisi pitänyt hallita jollain tavoin virhetilanne ja siitä palautuminen.

7. SERVERLESS-ARKKITEHTUURIN HYÖDYNTÄMINEN ASIAKASPROJEKTEISSA

Tässä luvussa pyritään Gofore Oy:n asiantuntijoiden haastattelujen perusteella tunnistetaan miten serverless-palveluita ja -arkkitehtuuria on hyödynnetty laajoissa asiakasprojekteissa. Havaintoja hyödynnetään yhdessä Luvun 6 havaintojen kanssa serverless-palveluiden hyötyjen ja haittojen analysoinnissa Luvussa 8.

7.1 Goforen taustat

Gofore Oy on Tampereella, Helsingissä ja Jyväskylässä toimiva vahvasti kasvava it-konsultointiyritys, jonka liikentoiminta kattaa laajasti eri osa-alueita mukaan lukien käytettävyystutkimuksen, ohjelmistokehityksen, palvelumuotoilun ja johdon konsultoinnin. [39]

Gofore on muun muassa virallinen Amazon Web Services Consulting Partner, Channel Partner ja Authorized Government Partner, sekä lisäksi Gofore on myös Microsoft Silver Cloud Partner [39]. Gofore on valittu valtionhallinnon AWS-teknologiaan pohjautuvien pilvikapasiteettipalvelujen toimittajaksi [40]. Goforella hyödynnetään lukuisissa projekteissa AWS-pilvipalveluita, ja lisäksi Gofore järjestää myös pilvikoulutuksia AWS-teknologioista.

7.2 Haastattelukysymysten määrittely

Haastattelujen perusteella on tärkeää tunnistaa millaisten projektien, tietojärjestelmien ja arkkitehtuurien osaksi serverless-palvelut soveltuvat. Lisäksi tavoitteena on selvittää millaisia haasteita ja ratkaisuja projekteissa on havaittu serverless-palveluiden käytöstä. Kysymykset muodostetaan näitä tavoitteita silmällä pitäen. Kysymykset toimivat pohjana haastattelutilanteisiin, joissa käydään läpi avoimesti erilaisia yksityiskohtia projekteista.

1. Projektin perustiedot, mistä projektissa tai projekteissa on ollut kyse?

2. Onko alusta lähtien ollut selvää, että projektissa voi hyödyntää serverless-palveluita?
3. Miten mahdollisuus serverless-palveluiden hyödyntämiseen tunnistettiin?
4. Millaisia haasteita käytössä tuli?
5. Millaiset hyödyt serverless-palveluiden hyödyntämisestä saatiin? Kehittäjän näkökulma? Tuotteen omistajan näkökulma?
6. Miten serverless-palveluiden ratkaisemat tarpeet olisi voitu toteuttaa muutoin?
7. Oletteko tehneet kustannusarvioita kehitys- ja käyttökustannuksista eri tavoin toteutetulla ratkaisulle?

Yllä listattujen kysymysten tarkoitus on tunnistaa erilaiset projektit joissa serverless-palveluita on käytetty, sekä missä vaiheessa ja miksi mahdollisuus niiden käyttöön on havaittu. Lisäksi tunnistetaan haasteet ja ratkaisut niiden käytössä sekä mahdollinen serverless-palveluiden kustannusten vertailu muihin ratkaisuihin nähden.

Haastatteluihin osallistui kaksi asiantuntijaa. Tarkemmin haastatteluja käsitellään luvuissa 7.3 ja 7.4.

7.3 Haastattelu: Ollikainen

Joni Ollikainen osallistui haastatteluun etänä verkon ylitse 23.4.2017. Ollikainen on toiminut Goforen asiakasprojektissa Fonectalle, jonka puitteissa hän on tehnyt AWS Lambdoilla pieniä osia isompaan järjestelmään.

Kysymys 1: Projektin perustiedot, mistä projektissa tai projekteissa on kyse?

Ollikainen on ollut toteuttamassa pilottiprojektia markkinoinnin automaatiosta asiakkaan hakupalvelujen käyttäjille. Projektin tarkoitus oli rakentaa integraatio, jonka avulla saadaan kirjautuneiden käyttäjien tekemien hakujen perusteella tehtyä kohdennettua suoramarkkinointia. Käyttäjien hakusanat kerätään tilastointipalvelusta ja markkinointikamppanjaan sopivista hakusanoista lähetetään viesti markkinoinninautomaatioon.

Kysymys 2: Onko alusta lähtien ollut selvää, että projektissa voi hyödyntää serverless-palveluita?

Ollikaisen mukaan serverless-palveluiden hyödyntäminen, tässä tapauksessa AWS Lambdojen, on ollut alusta asti yhtenä potentiaalisena toteutusvaihtoehtona.

Kysymys 3: Miten mahdollisuus serverless-palveluiden hyödyntämiseen tunnistettiin?

Yksinkertainen hakusanan vertailu markkinointilistaan on pieni operaatio, jota halutaan tehdä skaalautuvasti ja kustannustehokkaasti. Serverless-palvelut olivat ennestään tuttuja ja tämä integraatio vaikutti lähestulkoon malliesimerkiltä niiden käytölle.

Kysymys 4: Millaisia haasteita käytössä tuli?

Projektissa käytettiin AWS Lambda -palvelua Serverless Frameworkin avulla. Serverless Framework hyödyntää AWS CloudFormationia, joka puolestaan vaatii monenlaisia oikeuksia toimiakseen. Lisäksi käytössä oli jatkuvan integraation palvelin Jenkins, joka myös toimii AWS ympäristössä. Sekä Jenkins -palvelimen, että itse Lambda-funktion tarvitsemien oikeuksien dokumentointi oli heikkoa, joten toimivien oikeuksien löytämiseksi jouduttiin tekemään useita kokeiluja. Serverless Framework ei myöskään aina kertonut kovin kuvaavasti, minkä oikeuden puuttuminen aiheutti julkaisun epäonnistumisen.

Kysymys 5: Millaiset hyödyt serverless-palveluiden hyödyntämisestä saatiin? Kehittäjän näkökulma? Tuotteen omistajan näkökulma?

Tässä tapauksessa serverless-palvelut toivat merkittäviä kustannushyötyjä suhteessa virtuaaliisiin palvelinkoneisiin. Lisäksi AWS Lambdan nopea ja rajaton skaalautuminen ovat selkeitä etuja. Kehittäjän näkökulmasta en kokenut, että tässä projektissa serverless-palveluista saatiin mitään erityistä etua.

Kysymys 6: Miten serverless-palveluiden ratkaisemat tarpeet olisi voitu toteuttaa muutoin?

Tässä projektissa serverless-palvelu olisi voitu korvata autoskaalausryhmässä toimivilla virtuaalisilla palvelinkoneilla, tai toteuttamalla integraatio suoraan osaksi tilastointipalvelua.

Kysymys 7: Oletteko tehneet kustannusarvioita kehitys- ja käyttökustannuksista eri tavoin toteutetulla ratkaisulle?

AWS Lambda oli selkeästi edullisin valinta infrastruktuurin kulujen puolesta verrattuna virtuaalisiin palvelinkoneisiin. Tarkka ero kustannuksissa riippuu virtuaalikoneiden autoskaalausryhmän konfiguraatiosta. Yöllä ja viikonloppuisin integraatioon huomattavasti vähemmän liikennettä, joten autoskaalauksen toiminta vaikuttaisi ratkaisevimmin kuluihin. Esimerkiksi yöllä hiljaisimpaan aikaan tulisi pitää yhtä virtuaalikonetta päällä ja maksaa siitä, jotta palvelu pystyisi reagoimaan reaaliajassa mahdollisiin yksittäisiin viesteihin vaikka koneen koko kapasiteettia ei tarvittaisi. Erityisesti hiljaisina aikoina AWS Lambda säästää kuluissa.

7.4 Haastattelu: Voutilainen

Jari-Pekka Voutilainen osallistui haastatteluun 7.6.2017 Goforen Tampereen toimipisteessä. Voutilainen on ollut osallisena projektissa, joka ei ole Goforen julkinen referenssi, ja siitä syystä projektin ja asiakkaan nimi on poistettu haastattelun sisällöstä.

Kysymys 1: Projektin perustiedot, mistä projektissa tai projekteissa on kyse?

Voutilaisen projektin korkean tason päämäärä oli tuoda julkisesti käytettäväksi avointa dataa organisaation sisäisistä tietojärjestelmistä. Organisaation sisäverkkoon ei pääse käsiksi julkisesta verkosta, joten tarvitaan erillinen ratkaisu avoimen datan julkaisuun.

Organisaation intraverkon ja AWS-palveluiden väliin konfiguroitiin välityspalvelin SpringBoot-sovelluksena, johon on pääsy vain intraverkosta. Intraverkossa oleva palvelin kerää julkaistavat datat ja lähettää ne välityspalvelimelle. Välityspalvelin ohjaa liikenteen AWS-pilveen rakennettuun järjestelmään Api Gatewayn kautta, jolla tarjotaan rajapinnat datalle ja sen metadatalle erikseen. Rajapinnat kutsuvat eri Lambdoja, joista toinen tallentaa metadatan AWS RDS(Relational Database Service) -tietokantaan ja toinen datan

AWS S3 -palveluun.

Muut osat järjestelmästä eivät hyödynnä AWS Lambdoja, joten niitä ei käsitellä tarkemmin tässä yhteydessä.

Kysymys 2: Onko alusta lähtien ollut selvää, että projektissa voi hyödyntää serverless-palveluita?

Alunperin projektissa ei suunniteltu käytettävän Lambdoja. Vastaava ominaisuus oli toteutettu ensin myös Amazon EC2 -instanssina. Päätös Lambdan käyttöönotosta tehtiin tietämättä olemassa olevasta EC2-instanssin päälle rakennetusta ratkaisusta.

Kysymys 3: Miten mahdollisuus serverless-palveluiden hyödyntämiseen tunnistettiin?

Lambdojen käyttöön oli tutustuttu koulutusten ja itseopiskelun tasolla, joten mahdollisuus niiden hyödyntämiseen tunnistettiin helposti. Tarve palvelulle on satunnainen ja lyhytaikainen, joten Lambda soveltuu tähän erinomaisesti. Jatkuvasti käynnissä oleva palvelu tälläiseen tarpeeseen on turhan raskas ja tarpeettomasti käynnissä odottamassa viestejä.

Kysymys 4: Millaisia haasteita käytössä tuli?

AWS Api Gatewayn konfigurointi oli erittäin työlästä, siten että Lambdan vastaukset saadaan muunnettua halutuiksi HTTP-vastauksiksi Api Gatewaylta asiakassovellukselle. Nämä muunnokset määriteltiin Api Gatewayn selainkäyttöliittymässä Integration Response -osiossa. Kehitystyön aikana ei ollut vielä valittavissa Api Gatewaylle Simple Lambda HTTP-proxy -vaihtoehtoa, joka olisi yksinkertaistanut Lambdalle luotavan API:n konfigurointia. Näin ollen vastaavat asiat ja oikeat HTTP-vastaukset täytyi määritellä itse Api Gatewaylle. Kaikki palvelut määriteltiin käsin AWS konsolista, mutta jatkossa tälläisiin voitaisiin hyödyntää myös AWS CloudFormation -palvelua tai muuta infran määrittäviä työkaluja.

AWS Api Gateway on myös aina julkinen verkkoon, joten viestit täytyy autentikoida jollain tavoin. Tässä tapauksessa tarkistettiin API-avaimella oikeudet Api Gatewaylle.

Kysymys 5: Millaiset hyödyt serverless-palveluiden hyödyntämisestä saatiin? Kehittäjän näkökulma? Tuotteen omistajan näkökulma?

Kehittäjän kannalta Lambdan kehittäminen on erittäin helppoa ja intuitiivista. Pienen, muutaman kymmenen rivin Javascript-sovelluksen voi kehittää suoraan AWS Lambdan selainkäyttöliittymässä ja testata siellä ilman muita palveluita.

Muille sidosryhmille ei Lambdan käytöllä ollut tämän projektin puitteissa mitään merkitystä. Voutilainen arvioi sovelluksen ylläpidon olevan jatkossa yhtä helppoa tai helpompaa kuin perinteisen Springboot-sovelluksen ylläpitäminen EC2-instanssilla. Kokonaisuutta vertailemalla Lambdan ja Api Gatewayn tekniseen toteutukseen on helpompi tutustua kuin vastaavaan tarkoitukseen kehitettyyn Springboot-sovellukseen ympäristöineen.

Kysymys 6: Miten serverless-palveluiden ratkaisemat tarpeet olisi voitu toteuttaa muutoin?

Käyttämällä EC2-virtuaalikonetta ja kuormantasaajaa ottamaan vastaan viestejä välityspalvelimelta. Projektin puitteissa oli jo toteutettu Springboot sovellus tähän tarpeeseen ennen Lambdojen konfigurointia ja käyttöönottoa.

Kysymys 7: Oletteko tehneet kustannusarvioita kehitys- ja käyttökustannuksista eri tavoin toteutetulla ratkaisulle?

Tarkkoja lukuja ei ole saatavilla, mutta käytännössä Lambda ja Api Gateway maksavat joitain senttejä tässä käyttötarkoituksessa. Api Gatewayn läpi lähetetään viestejä ainoastaan itse hallinnoidulta välityspalvelimelta ja erittäin satunnaisesti. Api Gateway toimii ainoana herätteenä Lambdalle. Jos viestiliikenteen määrä kasvaa, kasvavat myös kustannukset lineaarisesti. Tässä käyttötapauksessa ennakoidaan kustannusten pysyvän kuitenkin erittäin alhaisina.

Vaihtoehtoisen ratkaisun, eli EC2-instanssin, kustannukset olisivat olleet hieman suuremmat, ja vaihtoehto olisi ollut liian järeä tähän tarpeeseen. Lambdasta kertyy kustannuksia ainoastaan silloin kun sitä kutsutaan, mutta EC2-instanssista kertyy vähäisesti kustannuksia jatkuvasti sen ollessa päällä.

Kehityskustannukset Lambdalle ovat samaa luokkaa kuin perinteisen virtuaalikoneen päälle rakennetun ratkaisun, jos molemmat teknologiat ovat entuudestaan tuttuja. Kuitenkin tässä tapauksessa oltiin jo ennen Lambdan käyttöönoton päätöstä toteutettu

EC2-instanssin päälle rakennettu ratkaisu, ja jälkikäteen arvioiden ei olisi toteutettu Lambdoilla samaa palvelua uudelleen, koska kustannussäästöt käytössä ovat niin pienet suhteessa kehitystyön hintaan.

7.5 Tulosten analysointi

Näiden kahden suppean haastattelun perusteella havaitaan FaaS-palveluiden, näissä tapauksissa AWS Lambdan, olevan hyödyllinen lisä suuressa ohjelmistoprojektissa ja Lambdojen olevan käytössä laajenevissa määrin. Etuina Lambdoille koettiin olevan helppo skaalautuvuus sekä suorituskyvyn että kulujen suhteen. Vaihtoehtona Lambdoille nähtiin perinteiset virtuaalikoneet, joille voidaan konfiguroida automaattiset monimutkaiset säännöt skaalautumisen suhteen. Lambdojen koettiin helpottavan skaalautumista.

Lambdojen konfigurointi koettiin osin haastavaksi, mutta toisaalta palveluiden havaittiin kehittyvän jatkuvasti ja siten tulevaisuuden olevan hyvä konfiguroinnin osalta lupaava. Lambdat myös nähdään hyödyllisenä vain, jos niiden kanssa voidaan hyödyntää muita palveluita herätteinä järkevällä tavalla. Kustannuksissa haastateltavat kokivat saavansa säästöjä kulutuspiikkien aikana, mutta toisaalta muistettiin että pienissä palveluissa infrastruktuurin kustannuksista ei voida saada suuria säästöjä verrattuna kehitystyön kustannuksiin.

8. SERVERLESS-ARKKITEHTUURIN HYÖDYNTÄMINEN OSANA OHJELMISTOJÄRJESTELMÄÄ

Tässä luvussa käsitellään hyötyjä ja haittoja serverless-palveluiden käytöstä erilaisissa tilanteissa, peilaamalla kokemuksia Afterwork-alert-sovelluksen muutosprojektista asiantuntijahaastatteluista välittyneisiin havaintoihin.

8.1 Hyödyt serverless-arkkitehtuurista

Serverless-arkkitehtuuria käyttämällä sovelluksesta tulee helposti skaalautuva, logiikaltaan yksinkertainen, kustannuksiltaan pieni ja sen käyttöä on helppo analysoida. Esimerkkiprojektina olleen Afterwork-alert-sovelluksen muuttaminen serverless-mallin mukaiseksi tuotti merkittäviä hyötyjä. Aiemmin sovellus tuotti jonkin verran kustannuksia Amazon EC2 -instanssista. Näistä kustannuksista päästiin täysin eroon käytön ollessa vähäistä. Käytettäviä Amazonin palveluita, eli AWS Lambda, AWS Api Gateway ja AWS Cloudwatch -palveluita voi käyttää pienessä mittakaavassa täysin ilmaiseksi. Kustannuksia tulee vasta joidenkin miljoonien kuukausittaisten kyselyiden jälkeen, joten tämänkaltaisessa käytössä esitetty sovellus ei tuota koskaan kuluja. Goforen asiantuntijoiden haastatteluista välittyi samanlainen oletus asiakasprojektien kustannusten osalta, mutta kuitenkin muistutettiin että suuri osa kustannuksista tulee kehitystyön perusteella. Yksinkertaiset tilattomat funktiot saattavat kuitenkin vaikuttaa myös positiivisesti ylläpitovaiheessa tuleviin kehityskustannuksiin. Pienestä ja yksinkertaisesta sovelluksesta on helpompi löytää virhe, kuin suuresta monoliittisesta kokonaisuudesta.

Kompromissit, joita muutosprojektissa jouduttiin tekemään, voidaan laskea osittain myös hyödyiksi. Niiden ansiosta sovelluslogiikka yksinkertaistui, mutta sovellus kuitenkin säilytti loppukäyttäjälle tarpeelliset toiminnallisuudet. Yksinkertainen sovelluslogiikka on helpompi ylläpitää kuin alkuperäinen monimutkaisempi sovelluslogiikka. Serverless-sovelluksena vältetään myös alkuperäisen sovelluksen ajoittaiset ongelmat WebSocket-yhteyden luonnissa ja ylläpidossa.

Pieniin osiin pilkotun sovelluksen analysointi ja metriikkojen kerääminen on myös AWS Lambdan kanssa yksinkertaista. On helppo tunnistaa mistä kustannukset syntyvät ja

tarpeen vaatiessa kehittää tunnistettuja ongelmakohtia. Sovelluksen skaalautuminen suurille käyttäjämäärille on myös erittäin yksinkertaista. AWS Lambdan varaan rakennettuna sovellus on jo oletuksena hyvin skaalautuva, toisin kuin EC2-instanssille rakennettuna.

8.2 Haitat Serverless-arkkitehtuurista

Serverless-sovelluksena toteutettu Afterwork-alert vaati kolmen eri palvelun yhdistämisen. Alkuperäiselle sovellukselle riitti käytännössä mikä tahansa virtuaalikoneena tarjottava pilvipalvelin. Kuitenkin erityisesti laajempiin järjestelmiin sovellettuna serverless-palvelut voivat kasvattaa kokonaisuuden monimutkaisuutta huomattavasti, koska useimmiten käytännössä tarvitsee yhdistellä useampia palveluita.

Serverless-alustan käyttö voi tuoda myös odottamattomia teknisiä rajoitteita sovellukselle tai käytettäville kolmannen osapuolen kirjastoille. Afterwork-alert-sovelluksen muutosprojektin yhteydessä tarvitsi kirjoittaa jonkin verran koodia uudelleen, koska projektin aikana AWS Lambda ei tukenut uusinta ECMAScript versiota. Tälläisen projektin puitteissa se ei ollut merkittävä ongelma, mutta tämä rajoittaa ohjelmistokehitystä laajemmissa projekteissa, joissa saattaa olla huomattavasti enemmän ulkoisia kirjastoja ja moderneja teknologioita.

Tilattomuus aiheuttaa myös haasteita sovelluksen arkkitehtuuria suunniteltaessa. Sovellus joutuu alustamaan kaikki tietokantayhteydet ja yhteydet ulkoisiin verkkopalveluihin jokaisella ajokerralla uudelleen. Tälläinen toiminta tuottaa viiveitä sovelluksen vasteajoissa loppukäyttäjän kannalta. Tästäkään syystä serverless-malli ei sovellu kaikkiin tilanteisiin. Afterwork-alert-sovelluksen tapauksessa päädyttiin muuttamaan sovelluslogiikkaa siten, että kaikesta tilallisuudesta päästiin eroon. Kaikissa järjestelmissä tämä ei varmasti ole mahdollista.

8.3 Serverless-palveluiden luontevat käyttötilanteet

Serverless-palvelut soveltuvat hyvin osaksi suuria järjestelmiä. Niiden avulla voidaan esimerkiksi mikropalveluarkkitehtuurissa toteuttaa yksinkertaisia palveluita. Tälläisiä voivat olla esimerkiksi palvelut, joiden käyttö ei ole jatkuvaa ja sovelluslogiikka on tilatonta. Esimerkkinä sopivasta serverless-mallin mukaisesti toteutetusta palvelusta voisi toimia kuvankäsittelypalvelu, jolle lähetetään lähdekuva, palvelu käsittelee kuvan ja palauttaa lopputuloksen kysyjälle tai tallentaa sen tietokantaan. Tälläinen sovellus ei vaadi tilallisuutta ja on helposti irrotettava osa suuremmasta tietojärjestelmästä.

Kaikkia tietojärjestelmiä ei kuitenkaan kannata pakottaa serverless-palveluiden varaan, vaan pitää ymmärtää serverless-palveluiden olevan parhaimmillaan vain osana tietojärjestelmiä. Helposti skaalautuvan luonteensa ansiosta serverless-palveluiden varaan on hyvä siirtää laskentaa muulta järjestelmältä. Tällöin muut osat tietojärjestelmästä eivät kärsi raskaista laskentaoperaatioista, jotka ovat siirretty serverless-alustalle. Serverless-palveluiden avulla on myös helppo tarjota integraatiota erilaisten palveluiden välille, mikäli palveluista saa soveltuvia herätteitä ilman jatkuvaa integraatiosovelluksen käynnissä olemista.

9. YHTEENVETO

Tässä diplomityössä käsiteltiin serverless-arkkitehtuuria toteuttamalla esimerkkisovelluksen muutosprojekti, asiantuntijahaastatteluja ja tarkastelemalla eri palveluntarjoajien palveluvalikoimia. Serverless-arkkitehtuurilla tarkoitetaan yleisesti arkkitehtuuria, jossa hyödynnetään jollain tavoin pilvipalvelussa olevaa laskentayksikköä, joka sammuu jokaisen suorituskerran jälkeen. Tämän työn puitteissa käsiteltiin erityisesti AWS Lambda -palvelua.

Esimerkkinä olleessa muutosprojektissa käsitelty Afterwork-alert-sovellus esiteltiin sekä alkuperäisessä muodossaan, että muutettuna AWS Lambda palvelun avulla suoritettavaksi. Muutosprojektin aikana tehtyjen havaintojen perusteella voidaan todeta serverless-palveluiden soveltuvan hyvin tämänkaltaiseen pieneen sovellukseen. Samalla tunnistettiin potentiaalisia haasteita, joita voi ilmetä siirrettäessä sovellusta serverless-palvelun varaan.

Tilallisuuteen perustuva sovelluslogiikka, mahdollisesti pidemmät vasteajat yksittäiselle suoritukselle ja vanhojen teknologioiden käyttö ovat lähes väistämättömiä heikkouksia serverless-palveluissa. Kuitenkin, jos sovellus ei kärsi näistä, serverless-palvelut voivat edesauttaa pilkkomaan sovellusta pienempiin osiin, ja sovelluksen skaalautuvuus isommille käyttömäärille on lähes automaattista. Serverless-palveluiden käyttö on tehokasta ja hyödyllistä osana suurempia tietojärjestelmäkokonaisuuksia.

Työssä vertailtiin myös suppeasti eri palveluntarjoajien palveluvalikoimia liittyen serverless-arkkitehtuuriin. Havaittiin, että kaikilla isoilla toimijoilla, eli Amazonilla, Googlella ja Microsoftilla, hinnoittelu on hyvin samankaltainen ja palveluvalikoimasta löytyy pääpiirteittäin rinnastettavat tuotteet. Lisäksi kolmannen osapuolen infrastruktuurin määrittelytyökalu Serverless Framework tukee jokaista näistä kolmesta. Serverless Framework yksinkertaistaa ja yhtenäistää eri palveluntarjoajien infrastruktuurin konfigurointia, mutta ei välttämättä tarjoa kaikkia alla olevien palveluntarjoajien tuotteiden ominaisuuksia.

Asiantuntijoiden haastattelujen perusteella serverless-palvelut ovat tulleet jo osaksi käytännön tietojärjestelmäprojekteja, ja niitä sovelletaan nimenomaan pienissä rooleissa

osana suurempia kokonaisuuksia.

LÄHTEET

[1] D. Beimborn, T. Miletzki, and S. Wenzel, "Platform as a service (PaaS)," *Business & Information Systems Engineering*, vol. 3, no. 6, pp. 381–384, 2011, [Online] Saatavilla: http://dx.doi.org/10.1007/s12599-011-0183-3.

[2] S. Walraven, E. Truyen, and W. Joosen, "Comparing PaaS offerings in light of SaaS development," *Computing*, vol. 96, no. 8, pp. 669–724, 2013, [Online] Saatavilla: http://dx.doi.org/10.1007/s00607-013-0346-9.

[3] "AWS EC2," https://aws.amazon.com/ec2, 2017, [Vierailtu 05-06-2017].

[4] M. Schöller, R. Bless, F. Pallas, J. Horneber, and P. Smith, "An architectural model for deploying critical infrastructure services in the cloud," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, Dec 2013, pp. 458–466.

[5] "Heroku application platform," http://www.heroku.com/home, [Vierailtu: 10.7.2017].

[6] R. Hastings, *Making the Most of the Cloud: How to Choose and Implement the Best Services for Your Library*. Scarecrow Press, 2013, [Online] Saatavilla: https://books.google.fi/books?id=H3FBAgAAQBAJ.

[7] S. Newman, *Building Microservices: Designing Fine-Grained Systems*, 1st ed. O'Reilly Media, February 2015.

[8] M. D. Hanson, "The client/server architecture," *Server Management*, p. 3, 2000.

[9] J. Spillner, "Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation," *ArXiv e-prints*, Mar. 2017.

[10] M. Roberts, "Serverless Architectures," https://martinfowler.com/articles/serverless.html, 2016, [Vierailtu 06-04-2017].

[11] "AWS Lambda," https://aws.amazon.com/lambda, 2017, [Vierailtu 01-05-2017].

[12] "AWS API Gateway," https://aws.amazon.com/api-gateway, 2017, [Vierailtu 01-05-2017].

[13] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, ser. MOTA '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:4, [Online] Saatavilla: http://doi.acm.org/10.1145/3007203.3007217.

[14] "AWS Lambda | Pricing," https://aws.amazon.com/lambda/pricing/, 2017, [Vierailtu 01-05-2017].

[15] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.

[16] J. Spillner, "Exploiting the Cloud Control Plane for Fun and Profit," *ArXiv e-prints*, Jan. 2017.

[17] "Add-ons – heroku elements," https://elements.heroku.com/addons, vierailtu: 11.4.2017.

[18] Swardley, "Amazon is eating the software (which is eating the world)," Nov 2016, [Vierailtu 07-07-2017]. [Online] Saatavilla: https://hackernoon.com/amazon-is-eating-the-software-which-is-eating-the-world-738888fb9e82.

[19] "AWS Serverless Multi-Tier Architectures," https://aws.amazon.com/whitepapershttps://aws.amazon.com/whitepapers/, 2015, [Vierailtu 26-05-2017].

[20] "The Context Object (Node.js) - AWS Lambda," http://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-context.html, 2017, [Vierailtu 07-07-2017].

[21] "Amazon Cloudwatch," https://aws.amazon.com/cloudwatch/, 2017, [Vierailtu 10-07-2017].

[22] "serverless.com," https://serverless.com/, 2017, [Vierailtu 28-04-2017].

[23] "AWS CloudFormation," https://aws.amazon.com/cloudformation, 2017, [Vierailtu 01-05-2017].

[24] "Azure Functions Overview," https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview, 2017, [Vierailtu 01-09-2017].

[25] "Pricing - Functions," https://azure.microsoft.com/en-us/pricing/details/functions/, 2017, [Vierailtu 01-09-2017].

[26] "Cloud Functions Overview - Cloud Functions Documentation - Google Cloud Platform," https://cloud.google.com/functions/docs/concepts/overview, 2017, [Vierailtu 01-09-2017].

[27] "Events and Triggers - Cloud Functions Documentation - Google Cloud Platform," https://cloud.google.com/functions/docs/concepts/events-triggers, 2017, [Vierailtu 01-09-2017].

[28] "Pricing - Cloud Functions Documentation - Google Cloud Platform," https://cloud.google.com/functions/pricing, 2017, [Vierailtu 01-09-2017].

[29] "node-untappd," https://www.npmjs.com/package/node-untappd, 2017, [Vierailtu 06-04-2017].

[30] "slack-node," https://www.npmjs.com/package/slack-node, 2017, [Vierailtu 06-04-2017].

[31] "lodash," https://www.npmjs.com/package/lodash, 2017, [Vierailtu 06-04-2017].

[32] "moment," https://www.npmjs.com/package/moment, 2017, [Vierailtu 06-04-2017].

[33] "ws," https://www.npmjs.com/package/ws, 2017, [Vierailtu 06-04-2017].

[34] "AWS Lambda | Document History," http://docs.aws.amazon.com/lambda/latest/dg/history.html, 2017, [Vierailtu 05-06-2017].

[35] N. Prusty, *Learning ECMAScript 6*. Packt Publishing Ltd, 2015, pp. 19–22.

[36] "Node.js Releases," https://nodejs.org/en/download/releases/, 2017, [Vierailtu 05-06-2017].

[37] "Slash Commands," https://api.slack.com/slash-commands, 2017, [Vierailtu 05-06-2017].

[38] "Use Amazon API Gateway Custom Authorizers," http://docs.aws.amazon.com/apigateway/latest/developerguide/use-custom-authorizer.html, 2017, [Vierailtu 06-06-2017].

[39] "Gofore," https://gofore.com, 2017, [Vierailtu 06-04-2017].

[40] "Valtionhallinnon pilvikapasiteettipalvelujen toimittajiksi Capgemini, Fujitsu ja Gofore," http://www.valtori.fi/fi-FI/Valtionhallinnon_pilvikapasiteettipalvel(5205), 2017, [Vierailtu 06-04-2017].